



Rec'd PCT/PTO 22 DEC 2003

PCT/GB 2003 / 0 0 2 7 7 8

10/519558



INVESTOR IN PEOPLE

The Patent Office

Concept House

Cardiff Road

Newport

South Wales

NP10 8QQ

REC'D 15 AUG 2003

WIPO

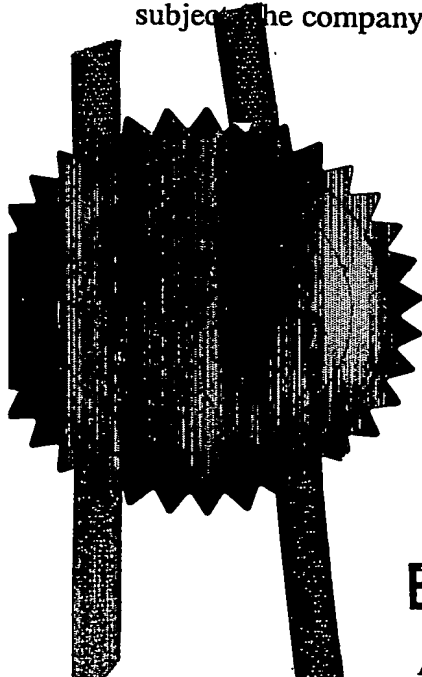
PCT

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.



Signed

Dated

11 July 2003

**PRIORITY  
DOCUMENT**

SUBMITTED OR TRANSMITTED IN  
COMPLIANCE WITH RULE 17.1(a) OR (b)

**Best Available Copy**

An Executive Agency of the Department of Trade and Industry

For Official use only



01786.02 E729611-1 D10092  
901/7795 0.00-0215033.2

Your reference **Translation (UK)**

**0215033.2**

**28 JUN 2002**

The  
**Patent  
Office**

Request for grant of a  
Patent

Form 1/77

Patents Act 1977

1 Title of invention

**Instruction Set Translation Method**

2. Applicant's details



First or only applicant

2a

If applying as a corporate body: Corporate Name

**Critical Blue Ltd**

Country

GB

2b

If applying as an individual or partnership  
Surname

Forenames

2c

Address

The Scottish Microelectronics Centre  
The Kings Buildings  
West Mains Road  
Edinburgh

UK Postcode EH9 3JF

Country GB

ADP Number 84 13775001

☐

Second applicant (if any)

2d

Corporate Name

Country

2e

Surname

Forenames

2f

Address

UK Postcode

Country

ADP Number

3 Address for service

Agent's Name

Origin Limited

Agent's Address

52 Muswell Hill Road  
London

Agent's postcode

N10 3JR

Agent's ADP  
Number

C03274

7270457002

4 Reference Number

Translation (UK)

5 Claiming an earlier application date

An earlier filing date is claimed:

Yes ☐

No ☒

Number of earlier  
application or patent number

Filing date

15 (4) (Divisional)

8(3)

12(6)

37(4)

☐☐☐☐

6 Declaration of priority

Country of filing

Priority Application Number

Filing Date

--	--	--

7 Inventorship

The applicant(s) are the sole inventors/joint inventors

Yes ☐

No ☒

8 Checklist

Continuation sheets

Claims 0

Description 77

Abstract 0

Drawings 0 *ov*

Priority Documents ~~Yes~~/No

Translations of Priority Documents ~~Yes~~/No

Patents Form 7/77 ~~Yes~~/No

Patents Form 9/77 ~~Yes~~/No

Patents Form 10/77 ~~Yes~~/No

9 Request

We request the grant of a patent on the basis of this application

Signed: *Origin Limited*  
(Origin Limited)

Date: *28 June 2002*

## Instruction Set Translation Method

### 1 Problem Description

For a new processor architecture to be successful a large quantity of support tools and software is required. Compilers must be made available that target the architecture along with the associated libraries and linker. A debugger is required to allow programs to be debugged while running on the architecture. Modern debuggers need to support symbolic level operation so that code can be executed with a view of the original source code. Software engineers expect an integrated development environment that ties the compiler and debugger tools into powerful GUI based environment. If software engineers cannot work in a familiar software environment with the tools they are used to using from day to day then this represents a significant barrier to the adoption of a new architecture. The development of such an environment and associated tools represents many tens of man years of development work even if existing compilers and tools can be retargeted to the new architecture.

In the CriticalBlue case the retargeting of existing compilers and debuggers is particularly problematic, as the processor has no fixed instruction set. Since CriticalBlue does not have a fixed instruction set it does support any kind of assembly language. Existing compiler and debug tools are designed to be targeted at a fixed architecture and require extensive modification to cope with a variable architecture. A fixed intermediate representation is needed to hide the variability of the architecture from existing software tools.

A particular tool trajectory has been chosen for CriticalBlue to minimise the amount of development effort and to promote adoption of the processor. The trajectory uses existing, and therefore familiar, software development tools. The fixed intermediate representation is in fact the machine code for an existing processor. Thus all the compiler tools and debug tools for that particular architecture can be used. The CriticalBlue code generator takes an executable for the processor as an input and produces an executable for a customized CriticalBlue processor as the output.

The translation must be able to take code generated for ARM and produce code for a particular CriticalBlue processor. This code must faithfully reproduce the same results as the original ARM code. However, the focus of the CriticalBlue architecture is to provide superior performance on particularly key parts of application. Even though this application code is expressed in ARM machine code the CriticalBlue code generator must be able to reorder and schedule the individual operations as required to make effective use of the innovative architectural features supported by CriticalBlue. Thus individual operations may be scheduled in a completely different order to that of the original ARM code.

The CriticalBlue environment must support the use of ARM debugging tools. In order to do this it must create the impression that the execution is equivalent to that of the individual ARM instructions. The engineer must be able to use all the features of the debugger. This includes the ability to set a breakpoint on any single ARM instruction. Program execution on CriticalBlue must halt as if it has reached that particular instruction. The set of values in memory and in registers must correspond to the values that would be expected if the code were running on a real ARM chip. The debugger may allow the program execution to be advanced at ARM instruction granularity. The effects of each ARM instruction must be precisely modeled in terms of their effect on register state and memory. The debugger also allows execution to be arbitrarily restarted from any individual ARM instruction in the entire program.

There is a conflicting requirement of allowing operations to be performed out of order with respect to the original ARM code. Moreover, in order to achieve high levels of parallelism on a CriticalBlue processor many aspects of the original ARM code must be optimised away, such as most accesses to the register file. Results may actually be generated in a completely different order to the way they were expressed in the original code. When single stepping or break pointing, however, operations need to be evaluated and results produced in the sequential order expressed in the original code.

A further issue is that the CriticalBlue processor needs to support instruction set extension. One of its most powerful features is that engineers can design new functional units and add them to the processor alongside any existing functional unit. This obviously creates a problem with the use of an existing instruction set in the tool flow for the processor. Such an instruction set is fixed and it is not possible to extend it.

## 2 Prior Art

There is a significant body of prior art in the area of instruction set translation. A number of academic and commercial systems have been built that allow binaries written for one architecture to be executed on another. One significant challenge is achieving high enough performance on the destination architecture. The exacting emulation of the particular idiosyncrasies of one architecture on another significantly degrades performance.

The simplest method is interpretive emulation. A soft CPU is built on the target architecture that is able to read and execute the instructions from the original architecture. Unfortunately this method is very slow and inefficient and is impractical for the type of embedded systems targeted by CriticalBlue. Moreover, this method does not allow the translated code to make use of the particular architectural features of the target.

The majority of recent research and commercialization in this field has been in the area of dynamic translation techniques. This method allows a very exact emulation of an architecture to be achieved while maintaining high performance levels. As code from the original architecture is encountered it is converted, at run time, into code for the target architecture using a dynamic code translator. The translated code can then be stored in a cache. The translated code can then be executed to produce the required results. If the same block of code needs to be executed then the translated version from the cache can be used again without the need to translate it again. In some systems an increasing amount of time is devoted to performing optimisations on a particular code sequence in the cache if it is frequently executed. Thus the run time system can target computationally optimisations on only frequently executed code. Dynamic translation systems can provide very exact emulations of architectures, even for events that are normally very difficult to handle in translation. For instance, self modifying code can be handled simply by flushing any affected code from the cache. Instructions that generate exceptions (such as memory accesses that generate a page miss) can also be handled and produce a machine state identical to that produced on a real machine of the original architecture. Breakpoints can be handled as exceptions so that if a breakpoint is encountered execution can be made to stop at a particular original instruction. Single stepping is achieved by producing translated code blocks consisting of a single original instruction. The most commercially evident dynamic translation system is that provided by Transmeta Inc. They have built a soft x86 processor that actually runs using a simple VLIW architecture, by utilisation of dynamic translation techniques. More recently Transitive Technologies have announced a more general technology that allows dynamic translation between a number of different processor architectures.

Dynamic translation is less suitable for embedded computing environments. Firstly, there is a significant memory overhead created by the translator itself and the size of the cache required in order to achieve good performance. Secondly, dynamic translation systems do not provide sufficiently deterministic behavior. Determinism is especially important for embedded real time environments. There is a significant start up delay while code from the application is translated into the cache. There may also be significant delays if an important block of code becomes evicted from the cache.

CriticalBlue employs a static translation technique. Static translation techniques have been used previously but have lost ground to dynamic techniques because they have a number of disadvantages. Fortunately these disadvantages are not of particular significance to the applications being targeted by CriticalBlue. Firstly, static techniques are unable to deal with self modifying code. Such code is rare in embedded environments. Code is often stored in ROM that cannot be modified anyway. Secondly, static translation is unable to provide support for precise exceptions. That is the translation cannot produce the exact results that would be expected if a source instruction causes an exception. This is significant if virtual memory needs to be supported as any memory access instruction might cause an exception if there is a page miss. This invokes a handler in the operating system that brings in the required page and continues execution after the memory access. Fortunately, CriticalBlue does not need to support virtual memory as the target applications are considerably lower level than those requiring such support. Thirdly, static translation techniques cannot handle indirect branches to arbitrary parts of a program. They need to have full knowledge of entry points within the code. Fortunately, such code is not generated by compilers or well written assembler programs. CriticalBlue processors are designed for programming from high level languages so this restriction does not pose a particular problem.

The final restriction is more significant. Prior art static translation systems have not been able to support debugging using the original instruction set. It is obviously imperative for CriticalBlue for it to support existing debuggers. The innovations in the CriticalBlue translation approach are primarily in the methods to allow such debug support.

Some existing configurable RISC processors (such as those supplied by Tensilica Inc and Arc Cores) have a facility to extend the instruction set. A number of unused operation codes are made available and are used to select an added instruction. The instruction execution logic has to be integrated into the pipeline of the processor in order to receive operands and write results back into the register file. This integration is more automatic in the case of the Tensilica solution. Both the Tensilica and Arc processors have their own instruction set and tool chain. The tools can be updated so that the new instruction can be accessed through the assembler using a user specified mnemonic. However, the compiler is not able to make use of a new instruction directly. The instruction is made available from a high level language by manually writing a wrapper function in assembler that invokes the required instruction. Since these solutions have a custom tool chain they do not suffer from the same problem as the CriticalBlue solution that uses a 3<sup>rd</sup> party, fixed, instruction set.

### 3 Summary of Contribution

The instruction set translation operates by converting each ARM instruction into a sequence of more basic operations to be executed on a CriticalBlue processor. All registers reads and writes are formed into separate operations. Thus a 3-address add operation is converted into operations to read the left and right operand registers, the add operation itself and finally an operation to write back the result to the register file.

Instructions with complicated addressing modes or that modify the condition codes result in longer sequences of basic operations.

A number of source instructions will generally be translated to form the contents of one individual strand within a region. In most cases a strand will contain all the operations from a basic block in the original code. The code will contain many reads and writes to the register file. To achieve high performance on the CriticalBlue processor (since it has limited register file connectivity) many of these accesses will be optimised away. For instance, if an operation generates a result that is written to the register file and subsequently read for use by a subsequent operation then code is optimised to write the data from the producer operation to the consumer. Register writes are preserved if they are writing data that will be subsequently used in other strands or regions in the program. Thus at the completion of strand execution the register state will be the same as on a real ARM chip for all registers holding values that might be subsequently used. Registers holding unneeded data may differ in value.

Thus if the program execution were to be stopped on a breakpoint on the boundary between strands then all the important register (and memory) state would be the same that observed on a real ARM processor. Of course, breakpoints can be set on any original instruction. Reducing the size of a strand to a single source instruction would dramatically reduce optimisation opportunities and thus the performance of the processor.

To overcome this problem, two versions of all regions generated for a CriticalBlue processor are produced. The main execution region is generated as discussed. A secondary, shadow, region is also produced. This contains the same translations for the same original instructions in each strand. However, in the shadow version, the individual register read and write operations are not optimised away. Moreover, the operations for each original instruction are performed in their original order. Each block of operations corresponding to an original instruction is bounded by a breakpoint check operation. This takes the original address of the instruction and compares it against all current breakpoints.

Break pointing on an individual original instruction is achieved as follows. A break point is set by specifying an original program instruction on which to halt. This is converted, using a table, into the address of a particular region and strand in the translated version of the program. The specified strand contains the translated form of that particular instruction. A breakpoint is set on the CriticalBlue architecture on that particular strand. Each region in the main code contains a conditional branch to the corresponding shadow region. If the breakpointed strand is about to be executed then that branch is taken to the same strand in the shadow code. At the point of the branch the registers and memory contents will correspond to that state of a real ARM processor at the first instruction in the strand. When the shadow code is entered it is executed until a match occurs between the original PC value and the breakpoint unit. In the shadow code all the effects of the original instruction are faithfully reproduced. When the actual breakpoint is reached the machine state will present that expected at the particular instruction. Thus execution proceeds at full speed until the strand containing the breakpoint instruction is encountered. Execution then proceeds more slowly on an instruction-by-instruction basis until the actual breakpoint instruction is reached.

Single stepping is performed by executing the shadow code while break pointing on every original instruction. At the end of each region a branch is made back to the main code. Thus when execution is restarted it initially executes the shadow code but a branch is made back to the main code as soon as the region execution is complete. Execution can

be restarted at any arbitrary instruction in the program by executing from the appropriate instruction in the shadow code.

The shadow code contains all the required operations to faithfully reproduce the behavior of the original ARM code. It is not heavily optimised like the main execution code. Thus the shadow code will be several times larger than the main execution code and, indeed, several times larger than the original ARM code. In an embedded system with limited memory this poses a problem. There is not sufficient memory space to store both the main execution code and the shadow code. The shadow code is only required when debugging is being performed, to support break points and single stepping. Thus during normal usage of the application only the main code is required and the code size is comparable to that of the original ARM version.

When the application needs to be debugged the shadow code can be held in memory connected externally to the system under debug. A slow bit serial interface can be used to connect this external shadow memory to the CriticalBlue processor. Since a serial interface is used the pin and silicon area overhead of the interface is very small. The disadvantage of a serial interface is that the access speed to the memory will be extremely slow. In fact the access speed could be two or three orders of magnitude slower than that to the main memory of the CriticalBlue processor. However, the shadow memory only needs to be accessed when a break point is encountered or single stepping is being performed. Moreover, only a small number of accesses need to be made during these events. Thus the performance degradation due to the slow access speed of shadow memory will not be noticeable and does not affect normal system performance.

Instruction set extension is supported by converting calls to particular software functions into an invocation of an extension hardware unit. This is achieved as follows. The engineer designs a hardware unit that performs the same operation as a particular software function. That is, it takes the same parameters and produces the same results as the code in its software equivalent. The advantage of the hardware version is that it will be able to produce the results more quickly. The engineer adds the function to a list that are implemented in hardware. The hardware function is given the same name as the equivalent software function. During translation, whenever a call to the software function is encountered it is converted into an invocation of the hardware unit. The parameters that would be passed to the software function are passed as the operands to the hardware unit. The results that would have been returned by the software function are obtained as the results from the hardware unit. In this way the effective instruction set of the CriticalBlue processor can be extended as required. The hardware functions can be accessed directly from a high level language just by calling the appropriate function. Moreover, this can be achieved without having to modify or extend the fixed instruction set of the source architecture.

## 4 Language Interface

### 4.1 Philosophy

The purpose of the software/hardware interface is to allow engineers to specify the boundary between hardware and software in a system. Software languages do not normally have to provide any facility for specifying that boundary. It is an intrinsic assumption that the underlying processor hardware will support a set of basic operations (such as addition, memory access etc.) that makes implementation of the software possible. All software is converted into a sequence of such operations by a compiler.

The CriticalBlue processor also has intrinsic hardware support for such operations. The support covers the hardware units required for implementation of all the instructions for

the processor machine code used as input to the CriticalBlue tools. However, the hardware units in the processor may be extended as required to implement more specialised functions for particular operations. Effectively, a CriticalBlue processor has a completely extensible instruction set.

Any particular software function may be annotated to indicate that it is actually implemented in hardware. Software functions are used as an abstraction for an operation actually implemented in hardware. Such functions must not have any side effects, such as the alteration of global variables or other areas of memory, since such operations cannot have a direct implementation in a hardware unit. Each function takes a number of parameters and produces one or more results based directly on those input parameters.

A function is indicated as having a hardware implementation by simply using a special prefix in its name. The CriticalBlue tools detect calls to such functions and automatically redirect them to uses of the associated hardware unit. Thus hardware extensibility is supported without the need to extend the software programming language. This methodology works from C and C++, as well as other high level languages and even assembly language programs.

A traditional processor uses a memory mapped, register based architecture for communication with hardware. This is somewhat foreign to most programmers so the interfaces are generally wrapped in a functional interface. A CriticalBlue system implements this functional style directly with no requirement for an intermediate register interface.

If C++ is being used then CriticalBlue can make direct use of the class abstraction. A class can be used to describe a particular hardware unit. The individual methods defined in the class correspond to operations that the hardware unit can perform.

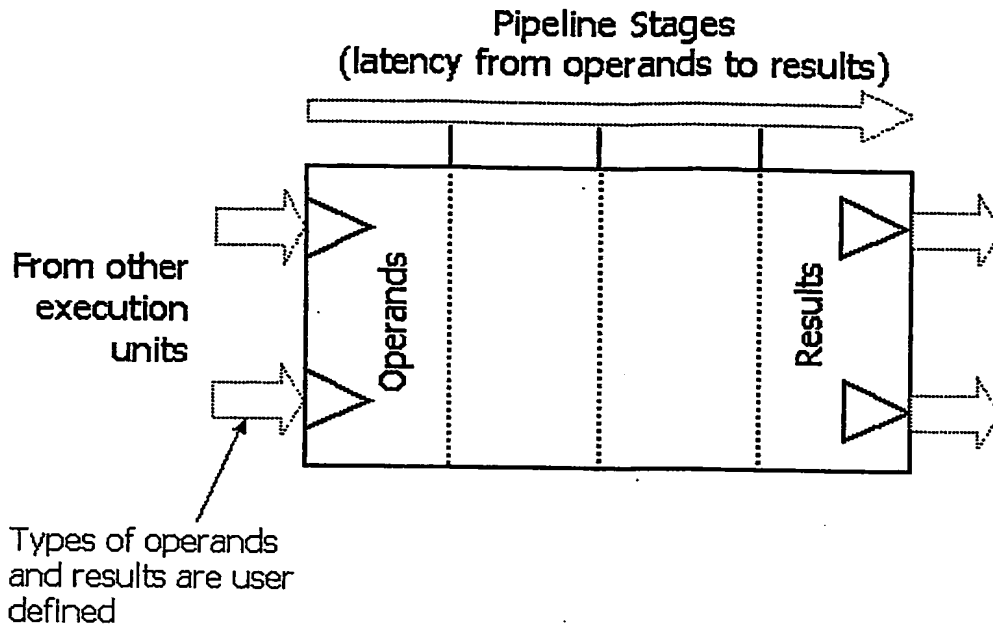
User defined hardware units do not have to be purely computational in nature. For instance, functions can be written to read and write to an array. This corresponds to an additional memory unit in the hardware of the processor. This is especially useful if extra memory units need to be defined to improve overall memory bandwidth for signal processing applications. Memory units can be defined of any size and width, allow arbitrarily complex memory architectures to be built directly from C/C++.

This methodology makes use of the power of high level languages to allow functions to be defined that effectively extend the language directly. There is no need for the explicit hardware/software interface that is generally characterized by architectures that need to be extended using a register access model. In effect the user is defining the instruction set of the processor by declaring functions to implement the instructions. Function calls map directly to the usage of those instructions.

## 4.2 Execution Unit Model

A single hardware execution unit may implement one or more individual functions. Each of these functions is termed a method of the unit. This corresponds to the terminology of a class encapsulation used in C++. Indeed, if C++ is used as the language to program a CriticalBlue processor then classes may be directly used to model a hardware unit, with the function members corresponding directly to these methods.

The diagram below shows the basic model of an execution unit:



The underlying model of an execution unit is as a synchronous, pipelined unit. This fits well with the computational model of units within a processor. A unit is able to accept a number of operands on a particular clock cycle and will produce a result a number of clock cycles later. This delay is referred to as the latency of the unit. The latency of the unit is represented by the vertical arrow on the left hand side of the diagram. It is expected that the unit is able to accept a new operation on every clock cycle. If necessary a blockage can be set for the unit that prevents it accepting another operation for a certain number of clock cycles after the last one.

The operands upon which the unit operates are shown as the input arrows at the top of the diagram. The types of these operands are completely user definable. They correspond to the parameters provided to the member functions defined by the unit.

An additional operand is automatically generated. This is the method parameter that indicates which individual method should be executed by the unit. It selects from one of a number of different methods supported by the unit. Different methods may operate on different sets of operands. The input operands to the unit represent the union of all parameters of the different method functions supported. Only those pertinent to the method being used need be set. The CriticalBlue tools handle this automatically.

The outputs from the execution unit are shown as the arrows at the bottom of the diagram. These correspond to either the return result from the methods or reference parameters. Reference parameters allow multiple results to be returned from a function. Thus a single operation can generate multiple results using this model. For instance, a division unit could return both the division result and the remainder as separate outputs.

Certain input operands and output results from an execution unit may be marked as being "dedicated". That is, they are connected to certain ports on other particular hardware units in the CriticalBlue processor. For instance, they might be connected to a control unit in the processor or to other user defined execution units. This mechanism allows hardware units to have an influence on the control flow of the processor. The mechanism may also be used to create custom communication paths between hardware units, some of which may be outside the CriticalBlue processor itself. Normally the input operands and output

results from a hardware unit are connected automatically as directed by the data flow in example programs. Designated operands and results are only connected to the specifically indicated ports on other hardware units.

The model for an execution unit is as a synchronous pipelined unit with a fixed latency. This is a good fit for most types of computational and logic units that will be used in the processor. If required, asynchronous operation can be supported by use of a software scheme to access the unit. One of the methods is used to initiate an operation. Another method may be used to read a flag that indicates when the operation has been completed. A software loop can be used to poll the unit until the result is ready. Further access methods may then be used to obtain the full results and status from the unit. If a polling scheme is too inefficient for a particular unit then an interrupt may be used. Any result bit from an execution unit may be routed directly as an interrupt source for a CriticalBlue processor. When the bit is asserted the CriticalBlue processor executes a predefined interrupt handler function.

### 4.3 Parameter Widths

The user is not confined to defining execution units that only operate on 32 bit (or perhaps 16 and 8 bit) data types. If required units that can be defined that operate on 4, 12 or 24 bit data types, or whatever is required. This allows much more efficient use of hardware resources. If a 4 bit operand is used then only 4 bits are physically routed to the execution unit.

The width of individual parameters is specified as a suffix to the name of the function. Any return parameter and then each of the function parameters has a number specified in the function name. This is a value between 1 and 32 and indicates the bit width of the operand. Wider actual parameters must be passed to the behavioral model. It must mask off higher bits not passed to the real hardware so that it generates the same result. For instance, a width of 20 bits may be specified for operands. The actual parameters to the function will be 32 bit integers.

If the user wishes to pass single parameters of larger than 32 bits to a hardware unit then it must be broken down into individual parameters of 32 bits or less. However, the full parameter may be encapsulated into a structure or class declaration. In C++, an inline member function for that class/structure can then be declared that passes the individual chunks of the parameter to a hardware implemented function. Since the member function is inlined the code executes efficiently but the subdivision of longer parameters is hidden within the inline function. Conversely, such a function may also be used to assemble multiple results into a single data item of more than 32 bits.

### 4.4 Behavioural Modeling

The software form of a function that is implemented in a hardware unit forms a behavioral model. That is, it describes the operation of the execution unit. The behavioral code is expected to produce exactly the same results that the real hardware would. Such code is executed on the host system during simulation. The code may access I/O or library functions that would not be present on a real CriticalBlue system. This allows the easy capture of trace information from execution units. Particular execution units might represent I/O units in the real system. These units can generate the appropriate stimulus required for the simulation.

The actual implementation of the hardware is generated separately from the behavioral implementation. Any development methodology may be employed as long as the behavioral model and hardware implementation remain equivalent. Normally, an

implementation is obtained by rewriting the software version into HDL. It can then be synthesized to generate an actual hardware implementation. Each execution unit only implements a fine grain component in the overall system so they are simple to verify.

Behavioural models are implemented as untimed. That is, the behavioral model is called synchronously and results are immediately returned. The simulation environment holds those results in temporary storage until the cycle on which the results should be available according to the latency of the execution unit.

If a timed behaviour model is required then input and result methods must be split. That is a timed method may only pass input parameters or receive output parameters (returns and/or reference parameters). A blockage can be set for the unit so that an output method is not called until at least a certain number of cycles after the last input method. Any function that is declared that has the prefix `hw_clock_` is automatically called on each clock cycle by the simulation. This allows the execution unit simulation to be advanced on cycles for which no method is explicitly called for the execution unit.

## 4.5 Use of Operator Overloading

The C++ language provides facilities to define functions allowing the built-in operators of the language to be use on user defined types. This allows very natural and concise usage of new types that are declared as part of an application. The operator functions take fixed parameters that are appropriate to the type of operation being performed.

Definition an operator overloading results in a function being created that is named from a combination of the operator involved and the name of the class (i.e. the type) that it operates upon. It is not possible to include the required `hw_` prefix into operator function directly so an overloaded function cannot be directly declared as being implemented in hardware. However, it is not desirable to do this in any case as operators rely on pointers to class objects being passed to them. Indirect memory accesses have to be used to obtain the data that is actually to be operated upon. A hardware implementation is unable to perform such an indirect access and must be passed the required data directly.

Fortunately the `inline` specifier of C++ allows overloaded operators to be defined that can efficiently use hardware implementations while maintaining the full abstraction of the language. The operators are declared as `inline` and their implementations call functions with a `hw_` prefix that actually operates on the data using custom hardware. Uses of the function become inlined so the hardware call is made directly and the overhead of a further function call is avoided.

The array access operator allows additional memory units to be declared and used in the architecture using the natural abstraction of array accessing. This is especially useful for signal processing style algorithms where memory bandwidth is critical. Some sets of operands on which the algorithm is operating can be declared using an overloaded array corresponding to a separate memory unit. Smart pointers can also be declared to access these additional memory units when they hold complex dynamic data structures.

## 4.6 Function Name Prefixes

Special identifier name prefixes are used to indicate that a function is implemented in hardware. A CriticalBlue prefix is simply the identifier `hw_` preceding the remainder of the name. If C++ is being used then the CriticalBlue tools are able to demangle the names to operate on the original identifier. Thus class methods may be used that have the prefix.

The prefix is primarily an indicator to the programmer. The definitive indication is if the identifier appears as a hardware implemented method in the hardware configuration file.

Taking the address of hardware functions is not permitted, as they cannot be called indirectly.

## 4.7 Fixed Function Names

A number of fixed identifier names are used for functions that have specific meanings in the system. Each such identifier includes the CriticalBlue prefix. All fixed identifiers are listed below:

### 4.7.1 Entry Point

This identifier `__start` indicates the entry point for the processor upon initialisation. Additional code is generated at the start of the entry function to set up the interrupt and monitor vectors specified using other fixed functions. Fast interrupt handlers are also locked down in the instruction buffer.

The entry function is likely to be partially implemented in assembler code for the host processor. This is because the code must also set up the initial value of the stack pointer. For systems having a main CPU then entry point call will be initiated by a remote function call from the main processor. This will pass information such as the location of the stack. If the system is operating without a main processor then the entry point is automatically executed after reset. The stack pointer will be set up using a literal value specified in the entry function.

If an uncached instruction buffer is being used then the first region for the function is placed at address 0 in the buffer. Execution commences from address 0 after reset. If a cached instruction buffer is being used then the first region for the function is placed in main memory at address 0. This region is fetched automatically into the instruction buffer after reset.

### 4.7.2 Software Interrupt Handler

This identifier `hw_swi` is the entry point for a software interrupt handler. It is called if a software trap instruction is executed. The trap instruction is translated into a call to the handler. Software interrupts are used to implement various low level functions. A CriticalBlue handler might be used to pass operating system service requests back to the main processor for handling.

On entry to the software interrupt handler certain registers are preserved using allocated registers in the register file. For an ARM processor, this implements the effect of the banked registers made available when entering a software interrupt handler.

### 4.7.3 Hardware Interrupt Handlers

Interrupt handlers may also be declared as functions. These are defined as ordinary functions but with special names to indicate their function. There are a fixed number of interrupt sources for the processor. Extra code is inserted around the function to preserve volatile registers in specially allocated registers within the register file.

All interrupts are responded to within a small and fixed number of cycles from the point of the request. Since the handler is called before the current region has finished executing current values have to be held in shadow output registers from the functional units. All functional units have such shadow registers in order to support interrupts. Multi-cycle units have run out queues. Results that entered the pipeline before the interrupt handler was entered are clocked out into a run out shadow queue. These may then be replayed on return from the interrupt handler.

A CriticalBlue processor supports two kinds of interrupts, fast and slow. The code for the fast interrupt handler functions is loaded into the instruction buffer and locked down permanently so there is no latency associated with loading the code from main memory. Fast interrupts may be used for high speed data acquisition purposes or other functions where there is close and time dependent interaction with the hardware. Slow interrupts do not have their handlers locked into the instruction buffer so there may be a delay while the required code is loaded. Nested interrupts are not permitted.

The naming convention is to use an identifier of the form `hw_fast_int_0` to `hw_fast_int_N` where `N` is the maximum number for fast interrupts. Similarly, slow interrupts are denoted by the names `hw_slow_int_0` to `hw_slow_int_N` where `N` is the maximum number. If a particular handler function is not defined then interrupts from the corresponding source are ignored.

The code defined for the handler is executed when the corresponding interrupt is received. Note that, unlike the code for hardware-implemented function, it is not behavioral code. It is actually executed in a real system.

#### **4.7.4 Monitor Interrupt Handler**

This is a special kind of interrupt handler that is used for processing interrupts generated by the debug communications unit. The special identifier `hw_monitor_int` is used.

A monitor handler operates in exactly the same manner as a slow interrupt except that the processor also enters monitor mode. This allows the handler to access functional units that are only accessible to the monitor such as the instruction predicate unit and the debug communications unit. A monitor interrupt is generated when new data is received from the debug communications unit. This is a debug link with a host computer used during debug.

#### **4.7.5 Monitor Stop Handler**

The monitor stop handler is entered when executing shadow code and no destination branch is selected. This event occurs when a breakpoint has been encountered. Shadow code is executed up to the breakpoint instruction in order to create the required machine state. The monitor stop handler is then entered. This can then communicate with the debug host computer using the debug communications interface. The special identifier `hw_monitor_stop` is used.

On entry to the function the volatile registers are saved to some special preservation registers in the register file. These registers are distinct from those used during an interrupt since the monitor may be entered within an interrupt routine for debug purposes.

#### **4.8 Example**

An example software definition for an execution unit definition is shown below:

class hw\_24BitALU { A C++ class definition groups a number of functions that are implemented in a single hardware unit. The class is identified as a hardware unit by the hw\_ prefix.

```
public:
    int hw_add24(int x, int y) {
        return (x + y) & 0xFFFFFF;
    }

    int hw_sub24(int x, int y) {
        return (x - y) & 0xFFFFFF;
    }
}
```

Method functions that should be performed directly in hardware. The prefix of hw\_ indicates the usage of hardware. The function implements a behavioural model for the corresponding hardware unit.

```
hw_24BitALU ALU;
```

A hardware unit may be declared in the usual way

```
function() {
    int a, b, c;

    a = ALU.hw_sub24(ALU.hw_add24(a, b), c);
}
```

Example usage of the functions. These are automatically converted into usage of the hardware unit. C++ operator overloading may also be applied to hardware function calls.

The example defines an execution unit for performing 24 bit addition and subtraction. The functions are declared as public member functions of a class. The functions are declared with the hw\_ prefix that indicates that it is the implementation of a hardware unit.

The example illustrates the power of the methodology. Within a few lines of code the user has defined a custom instruction for a processor. There is no need to resort to assembly language or any complex definition language. What is more, the program is completely standard C/C++ and is easily readable by any programmer.

## 5 Translation Principles

### 5.1 Architecture Choice

Third party architectures are chosen that are already popular in the application space being targeted by the CriticalBlue product. Thus the development tools and instruction set will already be familiar to the target engineering groups for the product, lowering the barrier to adoption. The initial intermediate representation to be used will be ARM code. Later versions are likely to support MIPS and PowerPC, other architectures popular in System-On-Chip design.

The translation must be able to take code generated for ARM and produce code for a particular CriticalBlue processor. This code must faithfully reproduce the same results as the original ARM code. However, the focus of the CriticalBlue architecture is to provide superior performance on particularly key parts of application. Even though this application code is expressed in ARM machine code the CriticalBlue code generator must be able to reorder and schedule the individual operations as required to make effective use of the innovative architectural features supported by CriticalBlue. Thus individual operations may be scheduled in a completely different order to that of the original ARM code.

### 5.2 Translation Technique

CriticalBlue employs a static translation technique. Static translation techniques have a number of limitations. Fortunately these limitations are not of particular significance to the

applications being targeted by CriticalBlue. Firstly, static techniques are unable to deal with self modifying code. Such code is rare in embedded environments. Code is often stored in ROM that cannot be modified anyway. Secondly, static translation is unable to provide support for precise exceptions. That is the translation cannot produce the exact results that would be expected if a source instruction causes an exception. This is significant if virtual memory needs to be supported as any memory access instruction might cause an exception if there is a page miss. This invokes a handler in the operating system that brings in the required page and continues execution after the memory access. Fortunately, CriticalBlue does not need to support virtual memory as the target applications are considerably lower level than those requiring such support. Thirdly, static translation techniques cannot handle indirect branches to arbitrary parts of a program. They need to have full knowledge of entry points within the code. Fortunately, such code is not generated by compilers or well written assembler programs. CriticalBlue processors are designed for programming from high level languages so this restriction does not pose a particular problem.

A final restriction is more significant. Prior art static translation systems have not been able to support debugging using the original instruction set. The CriticalBlue technology incorporates a number of innovations to overcome this restriction and allow use of existing debuggers.

### 5.3 Translation Operation

The instruction set translation operates by converting each ARM instruction into a sequence of more basic operations to be executed on a CriticalBlue processor. All registers reads and writes are formed into separate operations. Thus a 3-address add operation is converted into operations to read the left and right operand registers, the add operation itself and finally an operation to write back the result to the register file. Instructions with complicated addressing modes or that modify the condition codes result in longer sequences of basic operations.

A number of source instructions will generally be translated to form the contents of one individual strand within a region. In most cases a strand will contain all the operations from a basic block in the original code. The code will contain many reads and writes to the register file. To achieve high performance on the CriticalBlue processor (since it has limited register file connectivity) many of these accesses will be optimised away. For instance, if an operation generates a result that is written to the register file and subsequently read for use by a subsequent operation then code is optimised to write the data from the producer operation to the consumer. Register writes are preserved if they are writing data that will be subsequently used in other strands or regions in the program. Thus at the completion of strand execution the register state will be the same as on a real ARM chip for all registers holding values that might be subsequently used. Registers holding unneeded data may differ in value.

Thus if the program execution were to be stopped on a breakpoint on the boundary between strands then all the important register (and memory) state would be the same as that observed on a real ARM processor. Of course, breakpoints can be set on any original instruction. Reducing the size of a strand to a single source instruction would dramatically reduce optimisation opportunities and thus the performance of the processor.

### 5.4 Supporting Instruction Granularity Breakpoints

To overcome this problem, two versions of all regions generated for a CriticalBlue processor are produced. The main execution region is generated as discussed. A secondary, shadow, region is also produced. This contains the same translations for the same original instructions in each strand. However, in the shadow version, the individual

register read and write operations are not optimised away. Moreover, the operations for each original instruction are performed in their original order. Each block of operations corresponding to an original instruction is bounded by a breakpoint check operation. This takes the original address of the instruction and compares it against all current breakpoints.

Break pointing on an individual original instruction is achieved as follows. A break point is set by specifying an original program instruction on which to halt. This is converted, using a table, into the address of a particular region and strand in the translated version of the program. The specified strand contains the translated form of that particular instruction. A breakpoint is set on the CriticalBlue architecture on that particular strand. Each region in the main code contains a conditional branch to the corresponding shadow region. If the break pointed strand is about to be executed then that branch is taken to the same strand in the shadow code. At the point of the branch the registers and memory contents will correspond to that state of a real ARM processor at the first instruction in the strand. When the shadow code is entered it is executed until a match occurs between the original PC value and the breakpoint unit. In the shadow code all the effects of the original instruction are faithfully reproduced. When the actual breakpoint is reached the machine state will present that expected at the particular instruction. Thus execution proceeds at full speed until the strand containing the breakpoint instruction is encountered. Execution then proceeds more slowly on an instruction-by-instruction basis until the actual breakpoint instruction is reached.

Single stepping is performed by executing the shadow code while explicitly testing for a breakpoint on every original instruction. At the end of each region a branch is made back to the main code. Thus when execution is restarted it initially executes the shadow code but a branch is made back to the main code as soon as the region execution is complete. Execution can be restarted at any arbitrary instruction in the program by executing from the appropriate instruction in the shadow code.

## 5.5 Instruction Set Extension

Instruction set extension is supported by converting calls to particular software functions into an invocation of an extension hardware unit. This is achieved as follows. The engineer designs a hardware unit that performs the same operation as a particular software function. That is, it takes the same parameters and produces the same results as the code in its software equivalent. The advantage of the hardware version is that it will be able to produce the results more quickly. The engineer adds the function to a list that are implemented in hardware. The hardware function is given the same name as the equivalent software function. During translation, whenever a call to the software function is encountered it is converted into an invocation of the hardware unit. The parameters that would be passed to the software function are passed as the operands to the hardware unit. The results that would have been returned by the software function are obtained as the results from the hardware unit. In this way the effective instruction set of the CriticalBlue processor can be extended as required. The hardware functions can be accessed directly from a high level language just by calling the appropriate function. Moreover, this can be achieved without having to modify or extend the fixed instruction set of the source architecture.

## 5.6 Translation Operation

### 5.6.1 Philosophy

The static translation process occurs as a post-link operation. The intention is that the make script for the software development environment is augmented to call the MetaMapper after the linking is completed. If the software IDE does not support the calling

of a post-link operation then a script can be used that incorporates both the link and the call of MetaMapper.

Since MetaMapper is run after linker it operates on a complete executable. There are no unresolved references and the locations for all data sections are determined. No support is provided for any kind of dynamically linking, although such support is unusual in embedded development environments anyway.

The executable image provided to MetaMapper should not be stripped. All internal symbols should be retained as they are used to help locate the start of separate functions within the executable. Translation of stripped executables is possible but the efficiency of the translation caching system is substantially reduced.

The translation takes the executable image and generates a new executable image that contains the translated code. This process never modifies the size of the file containing the image. The MetaMapper only modifies words within the executable code sections of the file. Thus MetaMapper does not have to recreate the section structure of the file. The symbol table is unmodified by the process. However, symbols representing entry points will locate the address of the original code entry point. Since exactly the same format is retained for the executable image, the standard tools can be used to download the image to the target system. Moreover, the image can be read as normal by debuggers in order to support symbolic debug.

If code is being generated for a CriticalBlue processor with an uncached instruction buffer then a separate buffer image file is produced. This is a direct binary representation of the code that should be loaded into the instruction buffer on power up. It contains all the translated code. In this case the original code image is left largely unmodified except for the insertion of address links at the required points in the image to support indirect function calls and returns.

### 5.6.2 Code/Data Differentiation

The MetaMapper must be able to differentiate between code and data in the image. All data must be left unmodified while all code must be translated. In some image formats all code and data are output to separate sections in the executable file. These sections are marked with appropriate attributes to allow code and data to be easily differentiated. The code and data are mapped to different base locations in the address map. Nearly all embedded environments allow writable data to be held in a separate section from code since the code itself may be run directly from ROM (which obviously cannot support read/write data).

Some development environments mix code and read-only data in the same section. In this type of environment a mechanism must be available to differentiate the two types of words. The ARM development environment mixes code and read-only data. However, it generates special markers in the symbol table showing the transitions between code and data. These markers are read by MetaMapper to enable the translation process.

### 5.6.3 Translated Image Areas

#### 5.6.3.1 Main Translated Code

This is the main section of the executable that contained the original code and data. All the data sections are maintained without any modification. Address links are inserted into the code at required points to support indirect branches.

If code is being generated for a processor with a cached instruction buffer then the original executable code is overwritten by translated code. There is no requirement for

any correspondence between the original code and the translated code it is overwritten with. The code does not have to belong to the same function.

#### 5.6.3.2 Original Code Copy

A separate section of the executable is used to hold a copy of the original code image. This is not used as part of execution and only needs to be present when debugging is being performed. It may be held in shadow memory. Only the monitor program accesses this data. This is done when a request is made to read the main memory. Equivalent data items from this section are returned rather than the actual content of main memory to give the impression that the original code image is being executed. Thus disassemblies of the code in the image will show the original code.

The code copy is structured so that only locations that have been modified need to be stored. If a particular word is unmodified in the translated code image then data is obtained from the main memory image. This prevents obfuscation of bugs involving the illegal writing of read-only data. If code is being generated for a processor with an uncached instruction buffer then only the words overwritten by address links need to be stored in this section.

#### 5.6.3.3 Shadow Code

This section is used to hold translated shadow code and is only presented for processors using a cached instruction buffer. If the instruction buffer is uncached then the shadow code is generated as part of the binary image to be loaded into the instruction buffer at power up.

Shadow code is an unoptimised version of the main code that is present to allow single stepping at original instruction granularity. In general the shadow code is several times larger in size than the main code. The shadow code only needs to be present when debugging and can be located in shadow memory.

#### 5.6.3.4 Mapping Table

The mapping table is used to translate original code addresses to the equivalent position in the translated code. The monitor program uses this table if an update to the PC register is performed. This may occur if the user utilises the debugger to jump to a particular function in the program. The monitor must determine which location, in the translated image, it needs to branch to in order to create the equivalent behaviour. The table only covers executable addresses. Attempts to load the PC register with values outside of the executable portion of the image are ignored.

The mapping table may either specify an address in the main code or an address in the shadow code. If the address is in the shadow code then the monitor must use the instruction predicate unit to only commence execution once the required instruction has been reached. The breakpoint mechanism is used as part of this process.

The table is structured to make use of the linear nature of translated shadow code. Individual instructions can be located using the instruction predicate unit. Within shadow code, ascending blocks of instructions are allocated to ascending strands. Thus within a region the table only needs to list the number of consecutive instructions within each strand.

The mapping table only needs to be present when debugging and can be located in shadow memory.

### 5.6.4 Image Padding

Since MetaMapper does not modify the size of the executable, the original executable must be large enough to hold the translated code and all the other image areas required. For an ordinary executable image this will generally not be the case since the shadow code is several times the size of the original code.

To provide an executable of sufficient size it is generally padded prior to the link stage. Special object files are provided that are filled with unused data in order to pad the image to a sufficiently large size. Ranges of padding files are provided of differing sizes. If the image is not big enough then MetaMapper reports that as an error and suggests the correct padding file to use. If the image size is too big and significantly too much padding is provided then MetaMapper is able to inform the user so that they can optimise the size of the executable image if required. The padding files are marked with special symbols so that MetaMapper is able to identify them as padding space rather than required code or data.

The padding for areas that may be located in shadow memory are provided as a section with a specified start address. A specific address is used to denote the base of the shadow area.

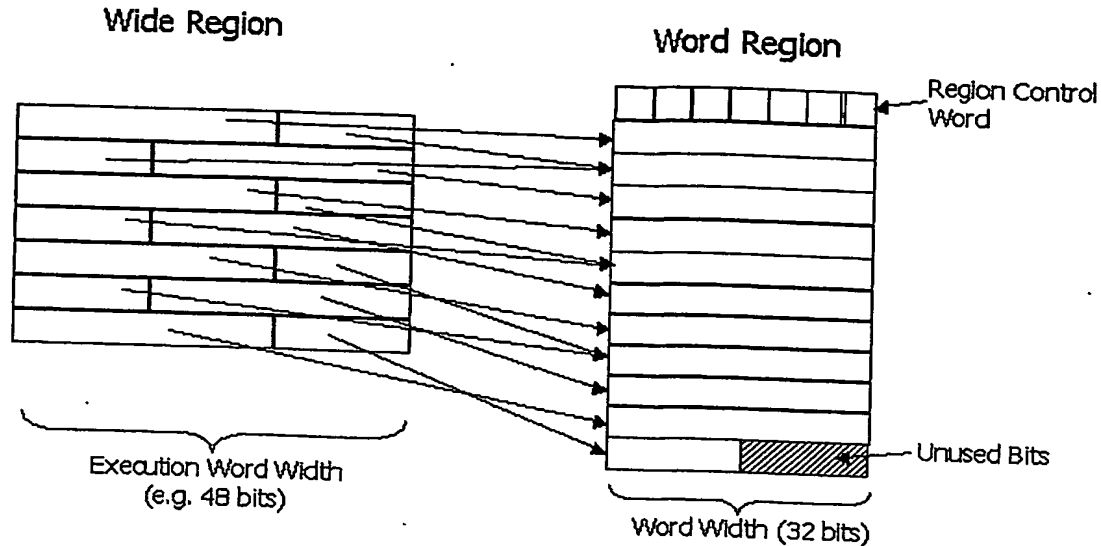
### 5.6.5 Wide Region to Word Region Mapping

The code that is to be executed on a CriticalBlue processor are formed into regions. These are contiguous blocks of execution words. The width of these words is user definable as the CriticalBlue architecture is able to support many different word widths.

In the case of an uncached instruction buffer processor, the translated regions are concatenated and used to generate a separate binary file. The data in the file is loaded into the instruction buffer on power up. The instruction buffer is the same width as the execution words for the processor.

If a cached instruction buffer is being used then the code within individual regions must be stored within the main memory of the processor. Regions are loaded into the instruction buffer on demand during execution. The width of the main memory is fixed at 32 bits so the wide form of the region must be compressed into a sequence of 32 bit words for storage in main memory.

This process is illustrated in the diagram below:



The width of the execution word is guaranteed to be 32 bits or wider. Thus the number of words in the 32 bit representation will be the same or greater than that in the original representation. Individual 32 bit words of data may be split across more than one execution word. The execution word width is guaranteed to be a byte multiple to simplify this process. As the region is loaded from memory into the instruction buffer it is reassembled into its wide form.

As illustrated there may be some unused bits within the last word of the narrower representation. An additional region control word is used to prefix the 32 bit wide representation. This contains information about the structure of the region as it is held in memory. The individual words may be split across fixed words in memory. The region control word also indicates the total length of the region, allowing the instruction buffer fill logic to determine how many words to transfer from main memory.

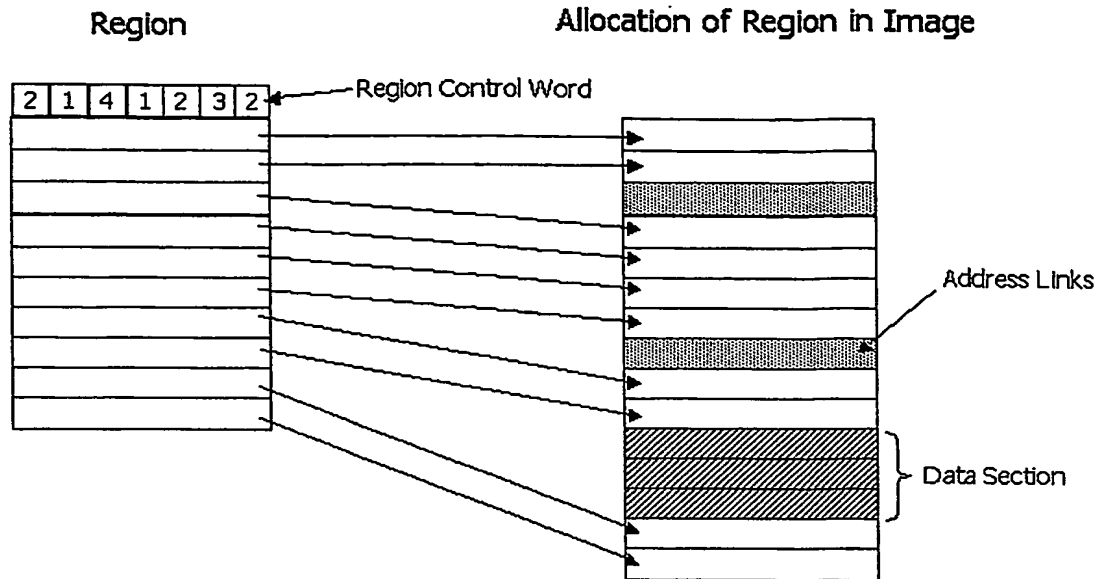
### 5.6.6 Word Region Address Allocation

If a processor with a cached instruction buffer is being used then translated regions must be allocated to addresses within the main memory. These addresses are used when specifying the start address of a region. If a region is to be executed and not present in the instruction buffer then it is copied into the instruction buffer for execution. Execution never occurs directly from main memory.

After narrowing from its wide representation, a word region is formed from a contiguous sequence of 32 bit words. In general these words are allocated to a contiguous sequence of words in main memory. However, in order to make efficient use of unused words in the translated image it must be possible to make use of fragmented space. The image can become highly fragmented due to the presence of address links and read-only data intermixed with code.

To allow placement of regions in a fragmented address space, each region is preceded by a region control word. This consists of a number of fields indicating the length of runs of contiguous words from the region interspersed with words that cannot be overwritten.

The following diagram illustrates an example region allocated to part of the image:



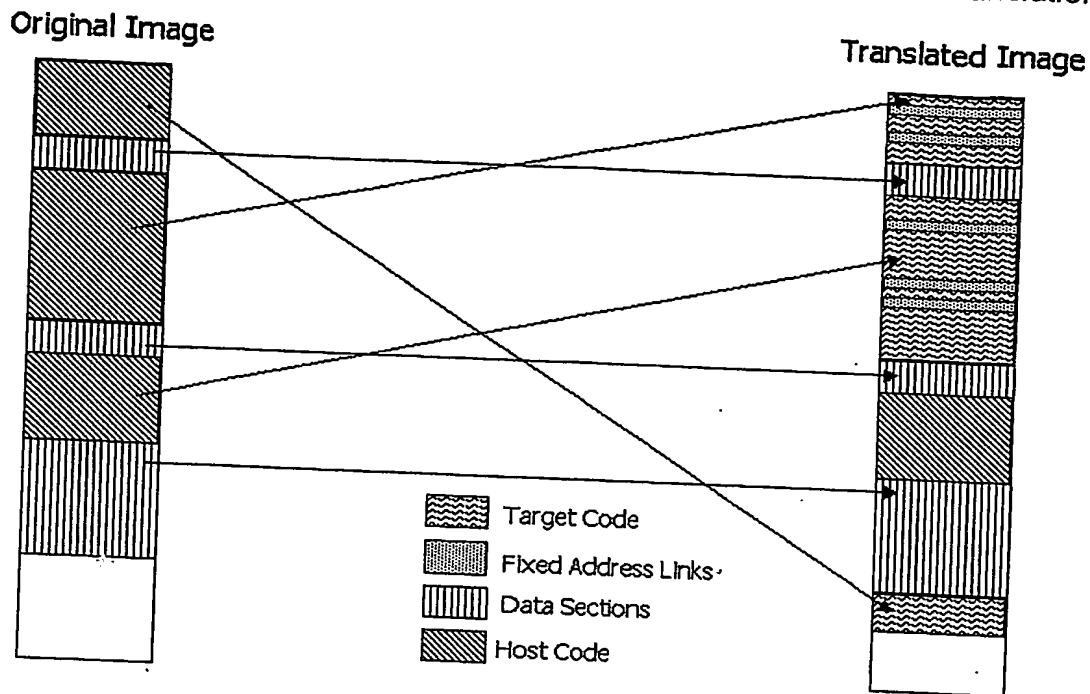
The mappings from the region words to locations in the address map are shown. The address map is fragmented with two address links and a block of three data words. The region words are flowed around these restrictions in order to make highly effective use of the space.

The region control word fields show the layout of the words in memory. The first field holds 2 and indicates the number of contiguous words before the first obstruction. The second field holds 1 and shows the number of words that must be skipped over. During the main memory to instruction buffer transfer this value is used to update the address pointer to skip over the required number of words. The third field holds 4 and shows the length of the next block of contiguous words from the region. The remaining fields continue to alternate between included and excluded blocks of words. If any field has a value of 0 then that indicates the end of the region has been reached.

### 5.6.7 Executable Map Allocation

The translated code, in the form of regions, must be allocated to the memory map. Address links, data sections and originally code that are not being translated needs to be preserved.

The following diagram illustrates an example allocation:



The position of data sections is unchanged, as is the data contained within them. Address links are placed at specific locations and cannot be moved. Any untranslated host code must also remain at the same location. The translated CriticalBlue code is then used to overwrite the original code in the image. There is no fixed relationship between host code and the origin of code that overwrites it. As illustrated in the diagram, host code from the start of the image may be translated into CriticalBlue code that is allocated at the end of the final image.

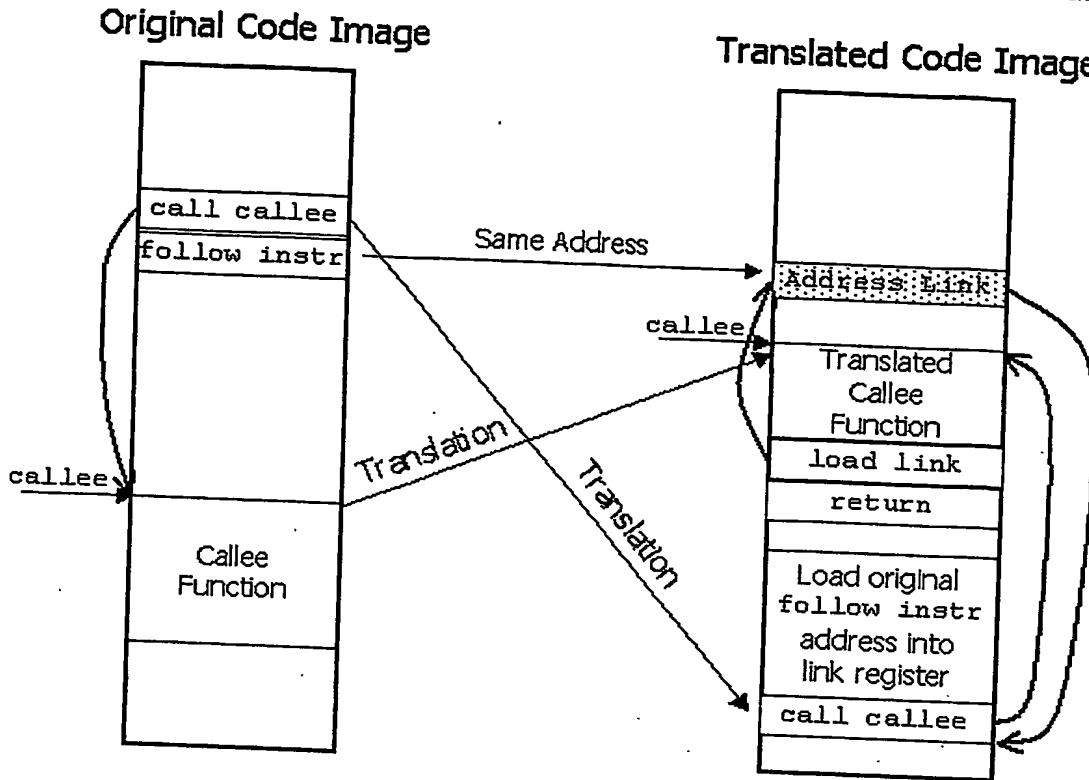
## 5.7 Address Linking

The address linking mechanism is a means for annotating the original code image to allow indirect branches to be supported. Such branches are required for returns and indirect function calls. The mechanism allows the original instruction addresses to be used for such operations.

### 5.7.1 Function Call Address Link

The function call address link mechanism allows the same return address that would be used in the original untranslated program to be used in the translated version. This is not normally possible because in general the calling instruction and, indeed, the function being called are located at a different address in the translated image. The return address is loaded into the link register by a call instruction in the original code image and this value is architecturally visible. The link register is preserved on the stack frame if the callee function makes any further calls. The debugger reads these preserved link values in order to generate a stack trace back and location the calling points represented on the stack. Thus to maintain compatibility with debuggers the original link address must be used.

The function call address link mechanism is illustrated below:



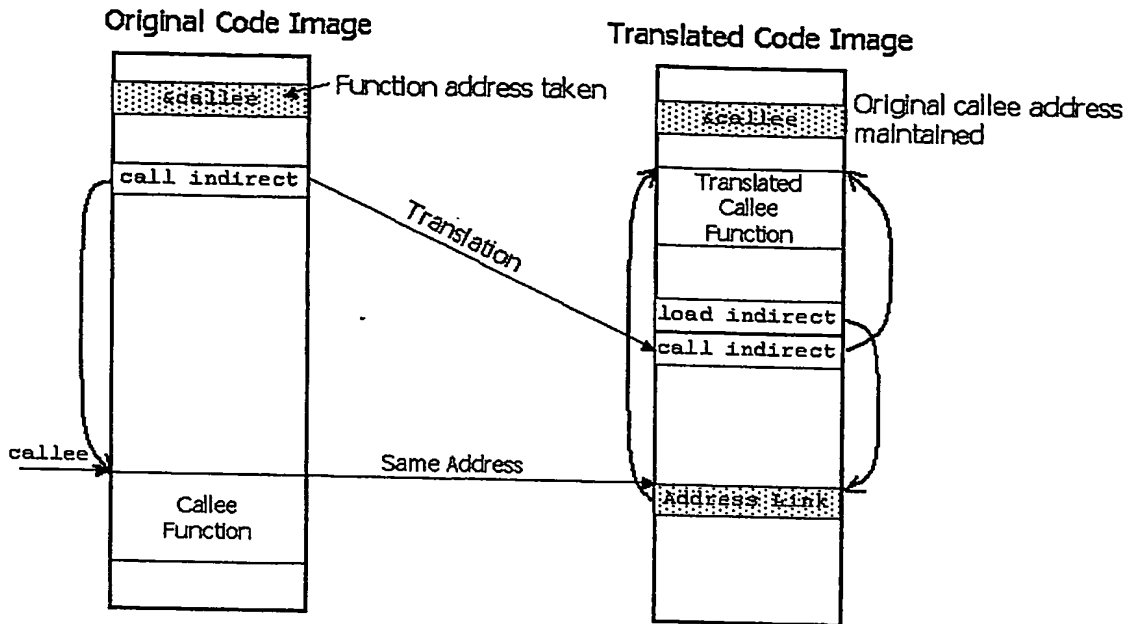
In the original code image a call is made to a callee function. The call loads the address of the instruction following the call into the link register. That is the address to which execution returns after the call. In the translated code image the location of the following instruction is used to hold an address link. An address link is simply a 32 bit data word strand to execute from that region. In this case the address link points to the operation following the call site in the translated code. The address link must be placed at the follow instruction address. Translated code is flowed around the address link as required. The translated version of the calling function explicitly loads the link register with the address of the follow instruction where the address link is now placed. In the callee function the return instruction (essentially an indirect branch to the link register) is converted to an indirect memory load and then an indirect branch. This loads the data held in the address link and then branches to that region. This continues execution after the translated call site.

This mechanism allows the original link addresses to be used and thus full compatibility maintained with debuggers for the host architecture. The only cost is the requirement to explicitly load the link register with an immediate address before a call and an extra indirect load at the point of a function return.

### 5.7.2 Indirect Call Address Link

The indirect call address link mechanism allows indirect calls to be made using the original, untranslated, addresses of functions. Indirect function calls are explicitly supported in most high level languages. In C++ all virtual functions are effectively implemented as indirect functions. The function addresses are held in a special compiler generated virtual function table that is associated with the class object using the virtual functions.

The diagram below illustrates how the mechanism works:



A data table somewhere in the code image provides the address of the function callee. At some point the code is able to perform an indirect function call using data that has been loaded from the table. The function address does not have to be loaded from the table directly before the indirect call. It may be loaded into a data structure some time before. Thus in general it is not possible to determine what set of functions any given indirect call might reach. The code analysis must assume that any indirect call can reach any addressed function anywhere in the code image.

In the translated code image the function address data item is maintained in exactly the same state. It points to the function address in the original code. At the original function address an address link data item is placed. This contains the address of the translated form of the function. The translated code must flow around this data item at a fixed address. Whenever there is an indirect function call in the code an indirect load is performed first. This obtains the contents of the address link. An indirect call is then made to the destination pointed to by the link. Thus all indirect function calls are made doubly indirect in order to reach the translated form of the function.

In general the code image is analysed and address links only placed at the start of functions that are detected as being addressed. However, a more simplistic analysis can place an address link at the start of all functions. This means that the analysis does not have to reliably detect all data items representing a function address. However, if this policy is utilised then it must be possible to detect the start address of all functions by other means (perhaps by assuming they always have a symbol and a preceded by return code).

## 6 Function List Creation

### 6.1 Requirement

The top level translation mechanism operates on a function granularity. All the code within a single function is converted into CriticalBlue code. Translation occurs at this relatively small granularity to allow the preservation translated functions from one invocation of the MetaMapper to the next. The processing time to translate an entire executable image of any size will be significant. This delay would be unacceptable in the normal edit, compile and debug cycle of software development. The compiler tools use a makefile system so that only modules in which code has changed are recompiled. The object files previously generated for other modules are simply linked into the updated executable.

The MetaMapper uses a caching system so that it only has to translate functions that have changed since the last invocation. In general only one or a small number of functions are changed on each development iteration.

### 6.2 Reference Counting

The first part of the task of creating a list of functions is producing a reference count table. This holds a count of the number of times each instruction is branched to from other parts of the code. The table is built by traversing all code within the entire executable. A linear map is created with one entry for each word in the executable image. Each entry holds a reference count of the number of branches that jump to that particular location. An entry may be marked with a maximum count, indicating that the word definitely represents the start of a function.

As the code is traversed, each time a branch is encountered the reference count for its destination is increased by one. If a call instruction is encountered then the reference count of its destination is marked with a maximum count. This ensures that all locations that are actually called definitely form the start of distinct functions from the analysis.

If possible all data is also traversed to detect addresses the addresses of functions. If a function is addressed then it is marked with a maximum count to ensure that it forms the start of a function in the analysis. In the case of executables by the ARM software development tools, all addresses of functions are marked with a special symbol marker. This allows such addressing to be reliably detected.

Function addresses may also be detected by examining retained fixup information. Linkers generally have an option to allow internal fixup information to be retained in the final executable. A function addressing has the characteristic of being relative to the base of a code rather than data section.

In the absence of these two possible mechanisms for detecting function addressing a third option is to use the symbol table. As long as internal symbols are retained all code addresses with an associated symbol can be marked as the start of a function. This analysis must be careful to discard compiler generated symbols that represent internal labels within functions.

### 6.3 Used Word Map

During the traversal of the executable image a used word map is created. This is a bit map, with one bit for each word in the image. If a bit is set then that indicates that the corresponding location is used and cannot be utilizing for holding translated code. All data words are marked as being used. Also, all locations requiring address links are marked as being used. Some individual instructions (such as software interrupt instructions with

immediate parameters) also have to be preserved in the executable image. They are also marked as being used.

## 6.4 Function List Creation

The function list is created using the reference count table. Entries that are marked with the maximum count definitely represent the start of functions. The function creation works by initially assuming that only entries marked with the maximum count are function entries and gradually splitting functions into smaller blocks if that assumption proves to be wrong.

A candidate function is delimited by two maximum count entries in the table. If all the control flow within the function is internal then the candidate function can be added to the function list. The code in the candidate function is traversed and this time branch destination reference counts are reduced. If a branch is encountered and the destination is within the candidate function then the reference count for the destination is reduced by one. If, at the end of the candidate function traversal, all the reference counts within the function are zero then all control flow is internal and there are no branches into the middle of the function. If there are non-zero counts then there is complex control flow resulting in such branches. In that case the candidate function is split into a number of smaller functions partitioned around the non-zero counts.

## 6.5 Translation Caching

The time to translate the whole of a sizeable executable image would be prohibitive. Thus a function level code caching mechanism is used from one invocation of MetaMapper to the next. This dramatically reduces the translation time for the majority of edit-compile-debug cycles where only a small proportion of the executable image changes. If the architecture of the target CriticalBlue processor changes then all code is invalid and the entire cache is cleared.

Before a new translation is made for a particular function a check is made to see if it is present in the translation cache. If so, and the original function code has not changed, then there is no requirement to translate it again. The associated translated form of the function is also stored in the cache and can be simply copied to the executable image.

The cache entry is identified by the symbols associated with its entry address. In most cases this is simply the functions name. Functions that do not have an associated entry symbol cannot be cached. However, such functions are unusual. An identifier is used because association with a particular address would not be practical, as functions move around as other functions in the executable are modified.

The binary image of the original function, used for comparison purposes, must also be invariant to these address changes. If a function includes a call to another function, as is commonly the case, then the change of destination addresses causes the binary to change. However, this should not cause the calling function to be re-translated as its actual implementation has not changed. The cached version includes structured data that points to all immediate fields holding addresses. These are ignored in the binary comparison process. Instead the symbolic destination is compared. As long as the same function name is being called then the function is considered to be unchanged.

## 7 Instruction Translation

### 7.1 Register Representation

A CriticalBlue processor has a central register file that is used to hold values that are written to registers in the host code. There is largely a one-to-one correspondence between these registers and those present in the host architecture.

For the main execution code, only those register values that are live at the conclusion of a strand need to be stored into the corresponding register. A register is live if it might be subsequently read in the program. Temporary uses of registers within a strand do not have to be reproduced in the register file. Thus the amount of register file traffic is very significantly reduced in comparison to the host architecture. This allows the register file to be synthesized alongside other execution units. It does not require special status or a large multiplicity of access ports. If necessary the logical registers may be split over a number of separate register file units to increase the amount of access bandwidth.

#### 7.1.1 Main Registers

There main registers are directly equivalent to those present in the host architecture. The majority of RISC host architectures have either 16 or 32 registers of 32 bit width. The same number of registers is present in the CriticalBlue register file.

Some architectures provide a special meaning to register 0, that is guaranteed to read as 0 and form a data sink for writes. In the CriticalBlue architecture, register 0 has no special meaning. Zero values are produced using an immediate unit and the effect of data write sinking is produced by simply not using the result from an operation.

A register must be provided that is used to hold PC values pointing to the original instructions. This register may be part of the main set of registers (as is the case with ARM7). The PC register is effectively live after each instruction. However, the main translated code does not keep this register up to date. Any read from the PC register is handled as a special case during the translation process. The shadow code keeps the PC register consistently up to date with the correct value from the host code after the completion of each host instruction.

#### 7.1.2 Condition Registers

CriticalBlue has separate registers for each of the individual condition flags. In other architectures these are usually combined as bit significant values into a single register. CriticalBlue uses separate registers because not all flags are updated by particular original instructions. CriticalBlue can emulate that behaviour by only updating the appropriate condition registers if the flags are live on exit from a strand.

The flag values are available as single bit output results from the arithmetic and logic units. If the condition code flag is not live then these output results can be read directly. If the associated flag is live then the result must be read and written to the appropriate condition code register. The connectivity between the result port and register file sign extends the result to 32 bits, so if the flag is set then all bits are 1 and if reset then all bits are 0.

Some condition code evaluations may require the reading of one or more flags and the logical combination of them in order to determine the condition value. If a read or write condition code register instruction needs to be translated from the host architecture then the individual bit significant flags of a condition code register are split/combined as required into the individual registers.

The individual condition code registers are as follows:

- ❑ **Carry Register:** Set if an operation resulted in a carry output.
- ❑ **Overflow Register:** Set if an operation resulted in an arithmetic overflow.
- ❑ **Negative Register:** Set if an operation resulted in a negative result (most significant bit set).
- ❑ **Zero Register:** Set if an operation resulted in a 0 result.

### 7.1.3 Special Registers

A number of additional special registers may be present for preserving the state of other registers during interrupts or entry to the monitor. These might be implemented as banked registers in the host architecture. However, the MetaMapper is able to generate special code on entry to and exit from the appropriate functions to preserve the required registers into addition special registers.

The registers associated with the software interrupt handler are ARM7 specific. If the host architecture is not ARM7 then other registers may be required.

The following is a list of the special registers required:

- ❑ **Software Link Register:** Used to hold the return link value from the software interrupt handler. This is transferred into main link register before the actual handler code is entered (but after the previous link register has been preserved).
- ❑ **Stack Preservation Register:** Used to preserve the stack pointer on entry to a software interrupt handler. A software interrupt handler can use a separate stack so the previous stack must be preserved.
- ❑ **Link Preservation Register:** Used to preserve the state of the link register during a software interrupt.
- ❑ **Condition Preservation Registers:** Used to preserve the state of the four condition code registers during a software interrupt.
- ❑ **Interrupt Preservation Registers:** Sufficient registers to preserve all volatile registers on entry to a hardware interrupt handler. The non-volatile registers are preserved using the normal prologue register saving code. The preservation set should also include registers for holding the four condition codes. It should also include the stack register, as a new stack pointer value will be loaded for the handler. This is required as the interrupt may occur during stack frame creation/deletion within the interrupted code. Only one set of preservation registers is required, as nested interrupts are not permitted.
- ❑ **Monitor Preservation Registers:** Sufficient registers to preserve all volatile registers on entry to the monitor. The non-volatile registers are preserved using the normal prologue register saving code. The preservation set should also include registers for holding the four condition codes. It should also include the stack register, as a new stack pointer value will be loaded for the monitor. This is required as the monitor may be entered during stack frame creation or deletion. Different preservation registers are used for monitor and interrupt entry so that the monitor can be entered to debug interrupt handler code.

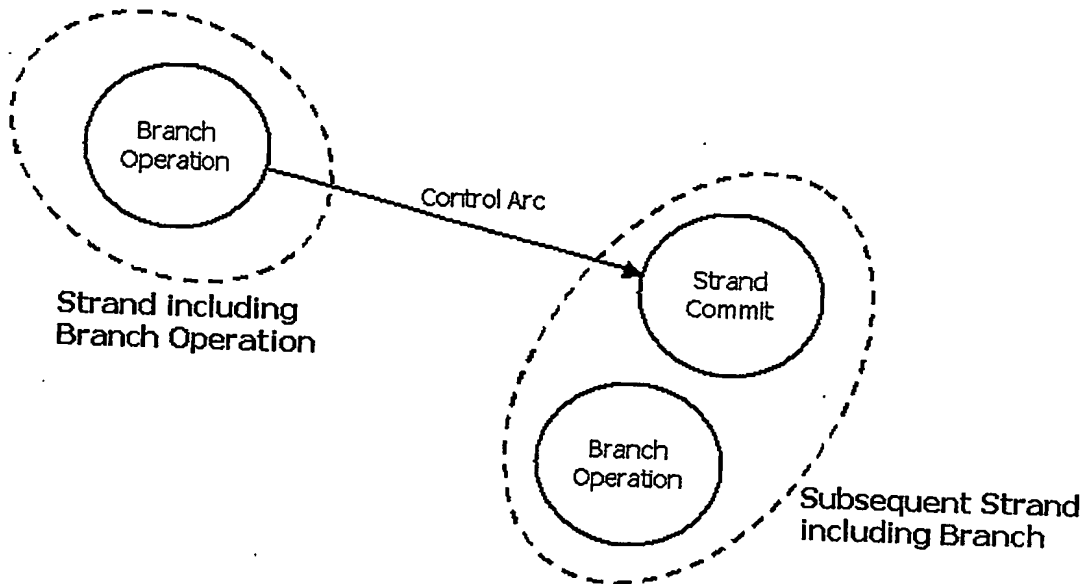
## 7.2 ARM Instruction Translation

This section describes how individual instructions in the host architecture are translated into sequences of operations for use by the CriticalBlue architecture. The descriptions are based on the mechanisms used for translating the ARM7 instruction set, but most of the techniques are widely applicable to RISC architectures in general

### 7.2.1 Branches

In the CriticalBlue architecture, all branches are delayed until the end of the region. That is, the branch destination can be issued at any point but the actual branch does not occur until the end of the region. Each strand may issue a single branch. Since the branch occurs at the end of the region the branch effectively occurs at the end of the strand. There cannot be any operations in a strand after a branch. Strands are always completed after issuing a branch since branch instructions delimit basic blocks.

Branches are resolved continuously so that if a branch is selected from a particular strand for execution then all higher numbered strands are automatically squashed. This is because they are not logically reached if an earlier branch is taken. This imposes a restriction on the issuing order of branches as illustrated in the diagram below:

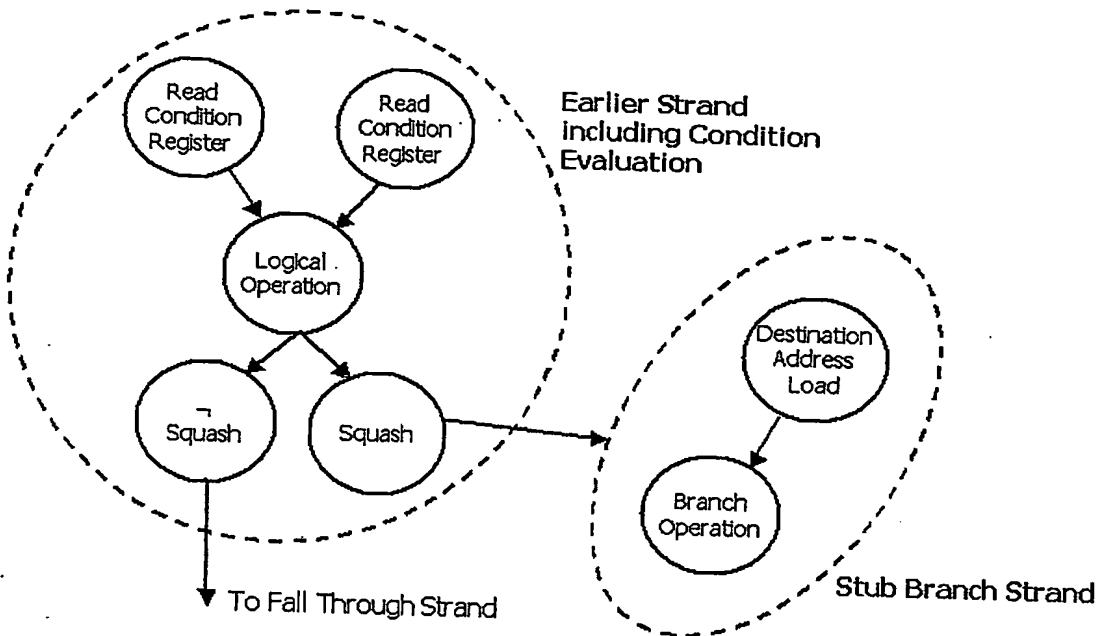


This shows an earlier strand that issues a branch. If there is a subsequent strand that issues a branch then its commit operation becomes a dependee of the earlier branch operation. This is because the earlier branch must be issued while the later strand is still in its speculative phase. If the earlier branch is actually taken then the later strand must not enter its committed phase. Note that the branch in the later strand does not have an imposed order so can actually be issued earlier than the first branch.

The branch itself is formed from two individual operations. Firstly there is an immediate load that sets the destination address along with the first strand to be executed at the branch. A wide immediate unit is used that also specifies the strand number that the branch is being issued within. The actual value is set via a fixup when the final binary is being written and the exact address has been determined. The second operation is the branch itself. The immediate value is passed to the branch unit. This value is then passed to the branch control unit.

If the branch being translated uses the same condition as that for the current strand then the branch can be issued within that strand. In general the condition for a strand is unconditional so this means that unconditional branches are issued in the same strand. However, if an individual instruction is conditional and then followed by a branch with the same condition (and no intermediate update of the condition codes) then that branch can be issued in the same strand as the previous instruction translation.

If the branch uses a different condition from that currently set for a strand then the condition must be evaluated and then branch put into a subsequent, stub, strand. This is illustrated on the diagram below:



A condition evaluation graph is shown in the first strand. In this example this involves reading two individual condition codes and forming a logical operation between them. The actual condition evaluation might be simpler or more complex depending upon the condition involved. The result of the evaluation is used by two squash operations issued in the earlier strand. One uses the true condition can is used to gate the branch and the other uses the inverse condition and is used to gate the fall through code translated into a later strand.

The branch itself is generated in a stub strand that is pointed to by one of the squash operations. Thus the execution of the branch is controlled by the squash. In many cases the branch will be to a destination that is subsequently shown to be within the same region. In that case the stub branch strand can be deleted.

### 7.2.2 Hardware Function Calls

If the host code contains a call to a function marked as being implemented in hardware then the call is translated into a use of the hardware unit. The software parameters are passed as the operands of the hardware operation. There will be a direct correspondence between the software function parameters and those that must be passed to the hardware unit.

The Application Binary Interface (ABI) of the host architecture will define how parameters must be passed to a function. This information is used during the translation process so that the locations of the parameters are known. In general the first few parameters are

passed in fixed registers and later parameters passed in fixed locations of the callers stack frame.

Code is generated to read each of the required parameters from the appropriate register. Later parameters are read from stack frame locations as required. These loaded parameters are then passed as operands to the hardware method. Some parameters may be marked in the hardware configuration file as discarded. These parameters are not passed. Other parameters may be marked as output parameters corresponding to pointers (or reference parameters) to hold results from the function. Again, these are not passed as operands.

If the software function provides a return result then this must be emulated from the hardware call. A function call result is normally returned in a particular fixed register. Code is generated to copy the result from the appropriate result port from the hardware unit to the register.

Other outputs from the function may have to be written to the address pointed to by an output parameter. Code is generated to obtain the parameter, representing the destination address, and generate a store of the result port to the address.

The wrapper code generated around the use of the hardware unit thus allows the hardware unit to provide the same behaviour as a software function implementation. Later optimisations performed on the CDFG can eliminate the write and subsequent read of parameter registers. This allows more efficient code that generates parameters and routes them directly to the operands of the hardware unit.

### 7.2.3 Software Function Calls

A software function is similar to a branch operation except that a link register is set prior to the call. The link register holds the return address from the call. In the host instruction the link register is implicitly set from the next PC value as part of the instruction operation.

In the translated CriticalBlue version the link register is loaded with an immediate value representing the address of the instruction following the call in the original program. This is the return location and points to an address link in the translated image. The immediate value is written to the link register prior to the actual call. The call is implemented as a load of the destination address and first strand, followed by a branch operation. The actual destination address is written to the code image later as a fixup when the destination address is resolved.

### 7.2.4 Software Interrupts

Software interrupts, or traps, is a special form of call that branch to a single destination or defined group of destinations. The software interrupt instruction contains an immediate field that is read by the software interrupt handler to initiate a certain type of operation. Entry to a software interrupt also generally changes the execution level. Software interrupts are usually used to implement a fixed interface to operating system functions.

In the case of the ARM7 architecture the software interrupt instruction causes execution to be vectored to the fixed address 8. In the case of the CriticalBlue architecture the software interrupt causes a call to be made to the special function `hw_software_interrupt`. If necessary a symbol of that name can be inserted at the start of the function implementing the software interrupt in an existing object file. Address 8 normally contains a single branch instruction that vectors execution to the actual implementation.

The address of the instruction after the software interrupt, in the host code, is loaded into a special alternative link register. A branch to the software interrupt handler is then

performed. Special code inserted into the prologue and epilogue of the software interrupt handler preserves the required registers and copies the alternative link register to the main link register.

### 7.2.5 Data Processing Instructions

The ARM architecture supports a number of data processing operations. All of these operations use a 3-address format where a left and right operand is specified along with a destination register. Some operations (such as compares and tests) do not actually cause a write-back to a register. A number of addressing modes are available for the right hand operand allowing immediate, register or shifted values to be specified. The instructions may optionally write to the condition code register.

The list of operations, their description and the implicit functional unit used to support them is shown in the following table:

OPERATION	DESCRIPTION	SUPPORTING UNIT	SUPPORTING METHOD
AND	Logical bit-wise AND of two 32 bit values	Logical	AND
EOR	Logical exclusive-or of two 32 bit values	Logical	Exclusive-OR
SUB	Subtraction of two 32 bit values.	Arithmetic	Subtract
RSB	Subtraction of two 32 bit values with operands reversed.	Arithmetic	Subtract
ADD	Addition of two 32 bits values.	Adder if no flags set else Arithmetic	Add
ADC	Addition with carry of two 32 bit values.	Arithmetic	Add with Carry
SBC	Subtraction with carry of two 32 bit values.	Arithmetic	Subtract with carry
RSC	Subtraction with carry of two 32 bit values with operands reversed.	Arithmetic	Subtract with carry
TST	Logical bit-wise AND of two 32 bit values but without write-back of results – flags used only.	Logical	AND
TEQ	Logical exclusive-OR of two 32 bit values but without write-back of results – flags used only.	Logical	Exclusive-OR
CMP	Subtraction of two 32 bit values but without write-back of results – flags used only.	Arithmetic	Subtract

CMN	Addition of two 32 bit values but without write-back of results – flags used only.	Arithmetic	Add
ORR	Logical bit-wise OR of two 32 bit values.	Logical	OR
MOV	Move operation. Operand loaded and written to destination.	N/A	N/A
BIC	Bit clear operation implemented using NOT of operand and then bit-wise logical AND	Logical	NOT + AND
MVN	Inverted move operation.	Logical	NOT

The individual instructions are translated into a number of separate operations on the CriticalBlue architecture. Firstly, operations are generated to load the right hand side operand. The sequence of operations required is dependent upon the addressing mode used and is described in the following sub-sections. An operation is then generated to read the left hand register operand. This is followed by the translated data processing operation itself. The majority of instructions map to a single data processing operation but some (such as bit-clear) map to two operations. If required then an operation is generated to write the result back to the destination register. If the instruction updates the condition codes then further operations are generated to update the affected condition code registers.

The single host instruction is translated into a number of individual operations. However, in general the later code optimisation phase will be able to eliminate many of the register file accesses to allow operands to be passed directly between the functional units. The shadow code always retains the full form of the translation so that a faithful emulation of the register state is maintained at host instruction granularity.

An immediate addition to the value of the PC register (as the left hand operand) is handled specially. Such an operation is generally used to calculate the address of a data item in a position independent manner. The full immediate value after addition is calculated and then a single operation is generated to load it via an immediate unit. The operation is marked as being fixed up so the address can change without forcing a re-translation of the whole function containing the instruction.

The handling of addressing modes is described below:

#### 7.2.5.1 Immediate Addressing

In this case the right hand operand is specified as an immediate value. This is divided into an 8 bit immediate field and a 4 bit immediate shift field. The shifted value is calculated and an immediate unit of appropriate width selected to load the value. The narrowest possible immediate unit is utilised in order to maximize code density.

Immediate units capable of loading full 32 bit values may not necessarily be available in the architecture. Thus the immediate value generated after the shift may be wider than can be loaded directly. In that case a two step process is used. The basic 8 bit immediate is loaded via an immediate unit. The availability of an immediate unit of at least 8 bits

width is guaranteed. The byte and bit shifter units are then used to shift the value appropriately to generate the required immediate value.

#### 7.2.5.2 Register Shift by Immediate Value

In this case a register value is shifted by an immediate amount. If the shift amount is zero then this addressing mode simply represents an operation performed on two registers. In that case the required register is simply read. If there is a non-zero shift value then the appropriate register value is read and then the byte and bit shifter are used as appropriate to generate the scaled operand.

#### 7.2.5.3 Register Shift by Register Value

In this case a register value is being shifted by a register amount. The initial value and shift amount must both be read from registers. Both the byte and bit shifters are used to perform the shifting. If a shift value of 0 is passed to either then they perform no operation. The shift units have special methods allowing them to extract the required shift amount from the appropriate bits in the shift amount operand. Thus they can operate directly with the shift amount specified in the register.

### 7.2.6 Multiply Instructions

The ARM architecture has a number of multiply operation variations. There are instructions to perform multiplies with both 32 bit and 64 bit results as well as multiply and accumulate variants. Signed and unsigned versions are also supported.

All multiply types use two register operands. These are read and the appropriate method executed on the multiply unit. If the result is 32 bits in size then only a single register write is required to put the result into the register file. If the result is 64 bits in size then two register writes are required from the lower and upper results ports of the multiplier unit. Finally, any condition code results are written to the appropriate condition code registers.

All multiply and accumulate variants are subdivided into separate multiply and then a subsequent addition operations. If there is strong data flow between the multiply output and the input to an arithmetic or adder unit then the architectural optimiser will provide strong connectivity between them. Thus a multiply-accumulate unit will emerge from the optimisation. If a 64 bit result multiply is being translated then the accumulator is 64 bits in width and two 32 bit additions (one with carry) are used to form the accumulation. If only a basic addition is required without condition code results then the adder unit is used rather than the arithmetic unit, as it is a smaller unit to replicate.

### 7.2.7 Memory Access Instructions

ARM supports memory access instructions with a number of addressing modes. The generation of the required code for the addressing mode is described at the end of this section. The ARM architecture supports both pre and post indexing of addressing modes. This allows an address to be automatically incremented or decremented as part of the access instruction without the requirement for additional address update instructions.

In the case of the CriticalBlue architecture these addressing modes and updates must be broken down into their constituent operations. The memory access unit uses the final computed address as its operand. In the case of pre-indexing the address is calculated and then written back to the base register if required. This address is then used for the access. In the case of post-indexing the address is simply formed from reading the base register. The access is performed and then the full address is calculated and written back to the base register.

If a full 32 bit memory access is being performed then the access itself only requires a single operation on the memory unit. Subword accesses are supported but require additional operations as the shifting is performed externally to the memory unit.

In the case of a load the shifting is performed after the access. The byte shifter is supplied with the operands of the loaded data and the loaded address. The byte shifter uses the lower 2 bits of the loaded address (which are ignored by the memory unit) to shift the data item appropriately. Different shifter methods are used for byte and half word data and for signed/unsigned variants. The data is moved to the least significant bytes of the result.

In the case of a store the shifting is performed before the access. The byte shifter is supplied with the operands of the data to be stored and the store address. The byte shifter uses the lower two bits of the store address (which are ignored by the memory unit) to shift the data to the correct position in the 32 bit word for writing. Bytes that are not being written will be undefined. Different shifter methods are used for byte and half word writes. The data size is also specified to the memory unit. Only the required bytes from the 32 bit word are written to memory. By explicitly shifting the data before the store, the memory unit does not need to have any internal multiplexers to direct data to the correct byte positions.

Once a memory access has been added to the CDFG, checks are made with previous memory accesses. Dependencies are added as required to ensure that the access is not scheduled earlier than aliased stores without appropriate compensation mechanisms being in place.

This following section describes the mechanisms used to generate code for an addressing mode. The number of addressing modes and their complexity is dependent upon the host architecture chosen. In general, RISC architectures have a rather limited range of addressing modes. This description is largely based on those available for the ARM7 architecture, which supports more modes than most comparative processor architectures.

#### **7.2.7.1 Zero Immediate Offset**

If the offset is 0 then code is simply generated to read the value of the base register.

#### **7.2.7.2 Non-Zero Immediate Offset**

In this case translated code is generated to load the offset using an immediate unit, load the value of the base register and then add the two values using an addition unit.

#### **7.2.7.3 PC Base Register with Immediate Offset**

Addressing using the PC register as a base is commonly used to access data sections via a position independent means. The value of PC at the point of execution is determined and the offset added to that value. This gives the address that will be generated by the addressing. This is loaded using an immediate unit wide enough to handle addresses.

#### **7.2.7.4 Register Offsets**

In this case we have two register offsets without any scaling applied to the right hand operand. We generate code to load the values of the registers on the left and right hand side. An addition operation is then generated to add the two values together.

In the case of ARM7 it is also possible to use a subtractive form of addressing where the right hand side is subtracted from the left hand side base address. In this case a subtraction rather than addition operation is generated.

#### 7.2.7.5 Scaled Offsets

In this case a register is used as the right hand side offset but is being scaled by an arbitrary shift value. In some architectures the range of shifts available is limited. The right hand register is read and the appropriate shifts are applied. The CriticalBlue architecture has separate byte and bit shift units. One or both of these units is used depending upon the shift value that needs to be applied. In some cases the carry output from the shift may have to be preserved.

#### 7.2.8 Block Memory Instructions

The block memory instructions allow multiple words to be loaded or stored to memory with a single host instruction. The behaviour of this instruction in the ARM architecture is unusual in that it does not conform to the general principles of RISC instruction implementation. It takes a variable number of clock cycles to execute depending upon the number of registers that need to be stored or loaded. The multiple word access instructions are commonly used in function prologues and epilogues to save and restore volatile registers on the stack frame. The ability to do this with a single instruction significantly improves the code density for the ARM architecture.

These block memory instructions are translated into multiple operations in the CriticalBlue architecture. The base register is read and then for each individual access (as determined by the register list in the host instruction) a memory operation is generated. An individual addition to the base address, using an immediate offset, is generated for each access. Individual offsets are generated rather than continually incrementing/decrementing a single address value. This allows the order of the loads/stores to be changed by the scheduler. This improved scheduling freedom allows the memory accesses to be more easily issued in parallel with other operations in a function.

Each individual memory operation is analysed for potential aliasing with preceding store operations. Dependencies are added to the CDFG as required to ensure that the correct operation ordering is maintained.

If the write-back bit is set in the instruction then an updated base register value is written. This is achieved by adding an immediate offset to the original base register value and then writing the result back to the same register.

#### 7.2.9 Memory Swap Instructions

Swap instructions combine the load and store of a byte or 32 bit word in a single atomic operation. This is used to implement semaphores where an automatic memory operation is required. CriticalBlue does not support automatic swap operations. However, other than atomicity the semantics of the swap instruction are translated. It results in both a load and store operation. Appropriate dependencies are generated to any potentially aliased preceding store operations.

#### 7.2.10 CPSR Access Instructions

The Current Program Status Register (CPSR) holds the state of condition code bits along with various other bits concerning the status of an ARM processor. Only the condition code bits are supported in the translated form of the code. Other bits within the CPSR are ignored.

The CriticalBlue architecture implements the individual condition code bits in separate registers to allow their independent read and update. This is done because certain instructions only update a subset of the condition code bits. It is only when explicit accesses to the CPSR are performed that the individual condition code bits need to be

formed into a single bit set in order to provide exact emulation of the architecture. Fortunately such explicit CPSR accesses are rare.

The sequence of operations generated is dependent on the type of CPSR access being performed.

#### **7.2.10.1 CPSR Read**

A CPSR read obtains the state of the CPSR register and writes it to a general purpose register. A CPSR read might be performed if the register needs to be preserved for any reason. A CPSR read is converted into a sequence of individual operations in the translated code. Firstly, a default CPSR value with all condition code bits reset is loaded using an immediate unit. The other bits in the register are assigned to reflect the standard user execution state. Each of the condition code registers for carry, negative, overflow and zero are then copied into the appropriate bit position in the value. This is simply done by masked off the appropriate bit from the condition code register (all bits in the registers are set to the value of the flag) and then ORing the value into the CPSR state being constructed. The final CPSR value can then be written to the appropriate general purpose register.

#### **7.2.10.2 CPSR Immediate Write**

A CPSR immediate write sets the condition code bits (and other bits within the CPSR) to specific values. Writes to the other bits are ignored but the writes to the condition code bits are translated into appropriate writes to the individual condition code registers. The immediate value is calculated. Operations are then generated to load a 1 bit immediate value for each condition code bit and write it to the appropriate register. The single immediate bit is sign extended to the full width of the register.

#### **7.2.10.3 CPSR Register Write**

A CPSR register write sets the condition code bits (and other bits within the CPSR) to values from a register operand. Writes to the other bits are ignored but the writes to the condition code bits are translated into appropriate writes to the individual condition code registers. Each of the individual condition code bits are masked off using an immediate value loading with an immediate unit. The zero result from the logical unit is then copied to the appropriate condition code register to set its new state.

### **7.2.11 Unsupported Instructions**

Certain ARM instructions cannot be translated to CriticalBlue architecture equivalents. An error is given if such an instruction is encountered during the translation process. An option to the MetaMapper allows the error to be downgraded to a warning and no code generated for the instruction (i.e. the instruction is ignored).

All instructions in the unused instruction space are unsupported along with all coprocessor access instructions.

## **7.3 Special Code Insertion**

In some cases additional code is inserted in the translation that was not present in the original host code. This is to support the use of standard functions for use as handlers without the need to resort to assembly language level coding. Additional instructions are inserted to save and restore additional registers. In the case of code insertion at the main entry point of the program, code is inserted to setup the interrupt handler vectors for the architecture.

### 7.3.1 Main Entry Point

Coe must be inserted to setup the interrupt vectors for the system. The inserted code is positioned before the translation of the very first host instruction at the entry point. The entry point itself may be denoted by two different means. If a symbol `hw_main` is defined then the function starting at that address is considered to be the main entry point. Thus the "main" function of the program can be declared as `hw_main` and any other entry code normally executed before the entry point is ignored. In the absence of such a symbol the first executable word in the code image is considered to be the entry point. This is normally situated at address 0.

The first operation issued is a disable interrupt operation to prevent interrupts being handled while the vectors are being changed. When the processor is first powered up interrupts are disabled anyway so there is no possibility of an interrupt occurring prior to the disable operation. A clear instruction buffer operation is also issued. This clears the contents of the instruction buffer including any locked down entries. This is required so that the lock down mechanism works correctly even if a branch is made to the entry point rather than execution after power up when the instruction buffer is empty anyway.

If the processor has a cached instruction buffer then all fast interrupt handlers must then be locked down into the buffer. This is achieved by calling each of the regions that are to be locked down in a special mode. A list is created of all the regions contained in the functions that are interrupt handlers. Each of these regions has a special strand that just performs a return without executing any of the operations in the region. This is always the last strand in the region. The branch to the region only executes the last strand. A special calling method is used that allocates the region being called to the start of the instruction buffer after any previously locked down entries. The calling region (containing the entry code) may be overwritten but it can be reloaded on return from the called region.

Once any fast interrupt handlers have been locked down the handler addresses can be set for both fast and slow interrupts. Each of the handler addresses is loaded using an immediate unit. The addresses are derived from the entry addresses of the translated handler functions. Each handler function has a special symbol name. Special branch methods are then used to set the vectors into the correct registers within the branch control unit.

Once all handler entry addresses have been loaded the interrupts are enabled. If the entry point was via the `hw_main` symbol then an operation is issued to load the initial value of the stack pointer. The address is specified as a command line option to the MetaMapper. The translated code for the first actual host instruction at the entry point is then executed.

### 7.3.2 Hardware Interrupt Handlers

Additional code is inserted at the start of the prologue to save the volatile registers into specially allocated registers in the register file. These registers are not normally saved by the function. However, for an interrupt function they need to be preserved as the interrupt can occur at any point during the execution of a function. The stack pointer register and the condition code registers are also preserved. The stack pointer is loaded with a new immediate value that is specified as a command line parameter to MetaMapper. The current stack cannot be used as the code may be manipulating the stack position at the point of the interrupt and there may be valid that below the stack pointer that would be overwritten. All the code inserted at the prologue is associated with the very first host instruction from the function.

Code to restore the preserved registers is inserted at each return point from the function. A single function may have a number of possible return points. The restore code is a mirror image of the preservation code and is associated with the instruction that modifies the PC register to cause the function return.

### 7.3.3 Software Interrupt Handler

Additional code is inserted at the start of the prologue to preserve a subset of registers into additional registers in the register file. This is to emulate the effect of entering a software interrupt handler on the ARM architecture where some registers are automatically preserved into another register file bank. Both the stack pointer and CPSR registers are preserved. Operations are inserted to copy the stack pointer and the individual condition code registers to special registers in the register file. The alternative link register (which holds the return address after the software interrupt) is then copied to the main link register. All the code inserted at the prologue is associated with the very first host instruction from the function.

Code to restore the preserved registers is inserted at each return point from the function. A single function may have a number of possible return points. The restore code is a mirror image of the preservation code and is associated with the instruction that modifies the PC register to cause the function return.

### 7.3.4 Monitor Handler

Additional code is inserted at the start of the prologue to save the volatile registers into specially allocated registers in the register file. These registers are not normally saved by the function. However, for a monitor function they need to be preserved as they may hold live values at the stopped point of program execution. The stack pointer register is also preserved. The stack pointer is loaded with a new immediate value that is specified as a command line parameter to MetaMapper. The current stack cannot be used as the code may be manipulating the stack position at the point of entry to the monitor and there may be valid that below the stack pointer that would be overwritten. All the code inserted at the prologue is associated with the very first host instruction from the function.

Code to restore the preserved registers is inserted at each return point from the function. A single function may have a number of possible return points. The restore code is a mirror image of the preservation code and is associated with the instruction that modifies the PC register to cause the function return.

## 8 Debug Environment

### 8.1 Overview

Before an application is ever run on real hardware it will have been tested in the CriticalBlue simulation environment. This allows full cycle and bit accurate testing. Stimulus and behavioural modelling code will be produced to emulate the physical environment that the application will be executed within. This process will allow the discovery of most major bugs in the application. Since the simulation runs natively using a C++ environment, the engineer is able to his or her favourite debugger and integrated development environment.

Of course, there are always likely to be application level bugs that only manifest themselves in the real hardware environment. To allow easy analysis of these, CriticalBlue supports a powerful debug environment. This is created from a combination of hardware structures embedded into every CriticalBlue processor and a software debugger capable of supporting source level debug.

## 8.2 Debug Chain

A standard debug port provides a connection between the CriticalBlue processor and the host debugging platform. This will utilise a standard serial data format. The debug port connects to a chain of serial connections between all the control and functional units within the processor. A serial protocol is used since high data speeds are not required and there is a need to minimise the area that the debug hardware occupies.

The debug chain allows individual registers in the processor to be read and modified under the control of the debugger. This allows the internal state of the processor to be interrogated and modified during the execution of a program. The overall processor clock is halted while these operations take place.

The debug chain allows access to any of the execution control registers in the control units. This includes the current program counter position and the various strand and branch control registers. The debug chain also provides access to the register holding the current instruction word being executed. The debugger can modify this register in order to inject particular instructions. This is the primary mechanism that allows the debugger to perform operations that change the state of the machine. For instance the user might type in a command in the debugger that modifies the state a memory location. The debugger converts it into a store instruction. It is injected into the instruction word and then executed by the processor.

Crucially, the debug chain also connects every single output register in all the functional units. This allows any item of data being held in the network to be read and modified. The debugger is able to gather the parameter values being passed to a particular functional unit for an operation. This allows it to show the actual parameters used for an operation. Registers may be modified in order to present particular parameter values to an operation injected into the processor pipeline.

The debug chain provides a powerful mechanism for debugging an application running on a CriticalBlue processor. Moreover, it does not require a significant area overhead and does not impact the timing of critical data and control flow paths.

## 8.3 Breakpoint Registers

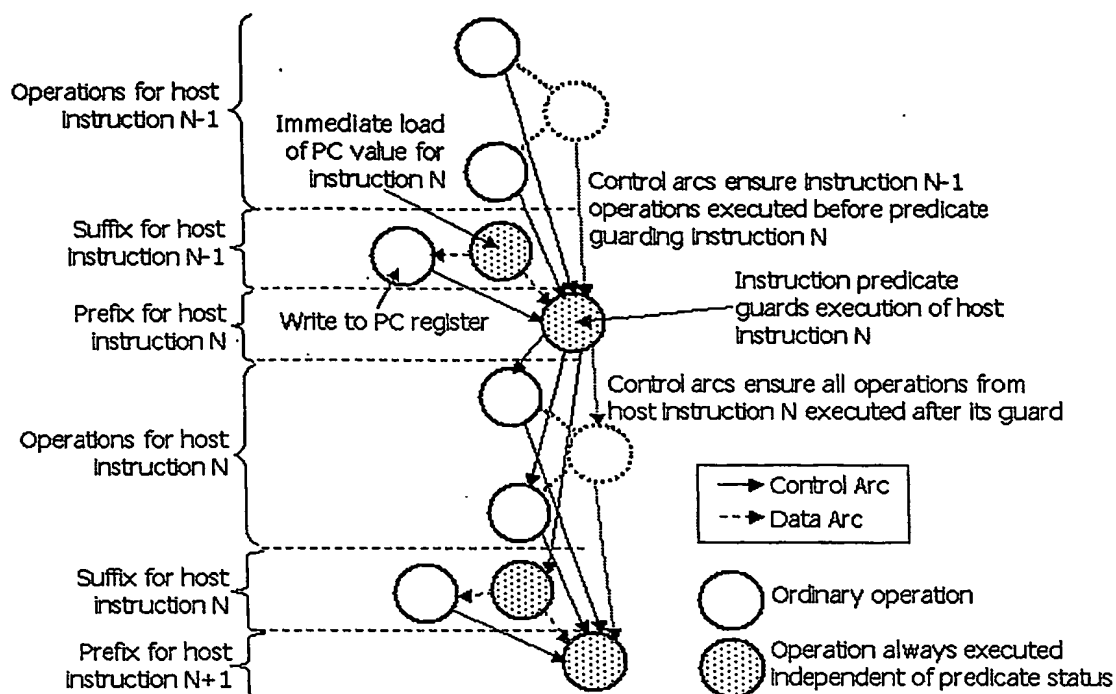
A CriticalBlue processor also contains a number of breakpoint registers. These cause the processor to halt if the breakpoint is reached. They are utilised if breakpoints are set in the debugger. A breakpoint register consists of a region start address and a strand number. Execution is halted if the particular strand in the region is executed. This allows breakpoints that halt the machine on the equivalent of a particular source line in the code. All later strands are squashed so execution is effectively stopped at the source line, even if instructions from later source lines have been issued.

## 8.4 Instruction Predicates

Instruction predicates are generated as part of the CDFG. They are used to delimit the translated code for a particular host instruction. A single host instruction is translated into a number of individual operations in the CDFG.

Instruction predicates are optimised out of the CDFG for the main code. However, they remain in the shadow code. They provide guards for the execution of operations related to a particular host instruction. This allows the single stepping of original host instructions when executing the shadow code.

The diagram below shows a segment of a CDFG with instruction predicates. This is typical of shadow code.



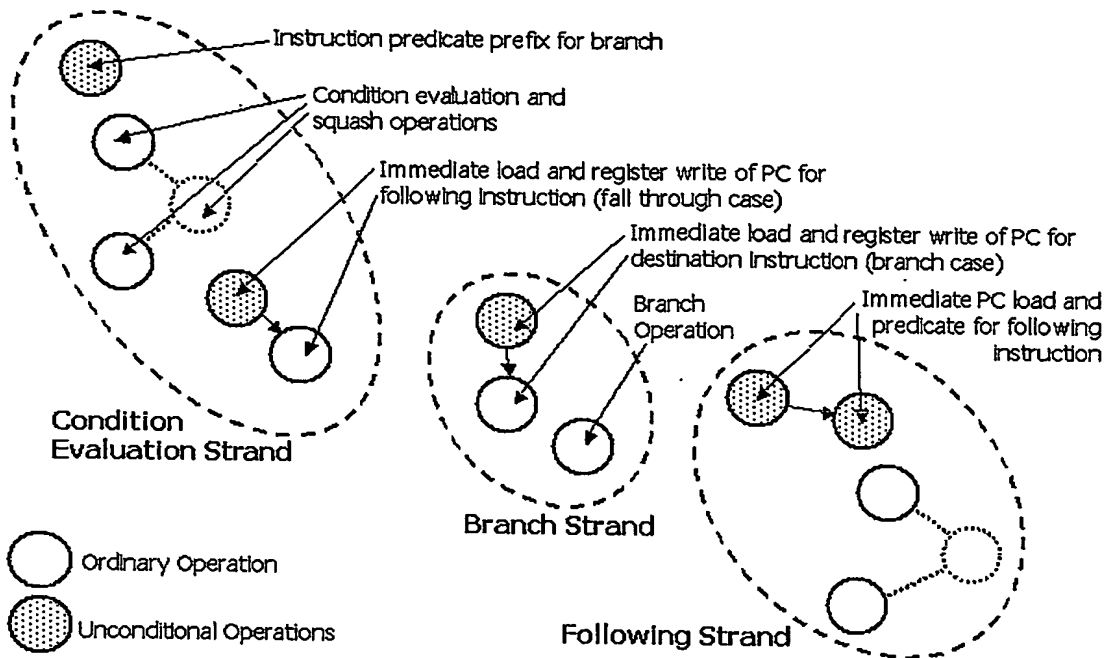
Each host instruction has a suffix consisting of two operations. Firstly, there is an immediate load of the original host instruction address of the following instruction. Immediate loads are always executed independently of the strand enable status. Secondly, the instruction address is written to the PC register in the register file. This emulates the behaviour of an instruction that sets the PC value of the next instruction to be executed. If single stepping of host instructions is being performed then this sets the correct address for the next instruction to be executed. The PC register write is executed in the strand of the instruction it is related to so is only executed if the operations for the instruction itself were executed.

Each host instruction also has a prefix consisting of an instruction predicate operation. The PC value loaded as part of the preceding suffix is provided as an operand to the instruction predicate unit. It compares the supplied address against breakpoint values held in internal registers. This determines whether the SEM should be enabled to allow execution of the following operations associated with the next instruction. Instruction predicate operations are always executed independently of the strand enable status. If the instruction is the first in the strand then there is no preceding suffix to obtain the instruction address from. In that case the immediate load is included as part of the first suffix.

As shown in the diagram, control arcs are added to serialise the operations for different host instructions in the CDFG. This ensures that any strand enable/disable caused by the instruction predicate unit only effects the relevant host instruction operations. All operations related to a particular host instruction are dependees of the prefix predicate for that instruction. This ensures the operations cannot migrate earlier than the predicate. The control arcs are labelled with the latency corresponding to the number of clock cycles required to update the SEM value after the execution of an instruction predicate. Also, the predicate for the following instruction is a dependee of all operations from the preceding instruction. This ensures that operations cannot migrate down into the operations associated with the following instruction. These control arcs occur within and between

strands. The arcs do cause a significant amount of serialisation of operations with a resultant adverse effect on the code density that can be achieved. However, these dependencies are only present for the shadow code. The shadow code is only required when debugging needs to be supported.

In the case of a conditional branch operation, the operations associated with the instruction may load of two successor PC values depending on whether the branch is taken or not. This is illustrated in the diagram below. Note that the diagram does not show the control dependencies between operations required to serialise their execution.



The first strand holds the instruction predicate for the branch along with the operations to evaluate the branch condition. The operations also include squashes to control the execution of a subsequent strand in which the branch is held. The first strand also contains operations to load the PC value of the subsequent instruction after the branch. These operations are always executed but may be subsequently overwritten by a branch destination PC value if the branch is actually taken.

The following strand containing the branch loads PC with the destination for the branch. The strand also contains the branch itself. The operations within the strand are only executed if the branch is being taken.

Finally, the following strand contains the operations associated with the instruction after the branch. This strand is only executed if the branch is not taken. The host instruction address is loaded and passed to the instruction predicate unit to guard execution of the operations. The PC value is explicitly loaded since the instruction prefix is at the start of a new strand.

## 8.5 Shadow Code

Shadow code is generated from the original source instructions. It represents a less efficient translation than the main code but provides support for instruction level debug. It

allows the machine state that would be generated by the original binary code to be reproduced to on a CriticalBlue processor.

When generating shadow code a number of additional operations are added into the CDFG that are not present for the main code CDFG. These additional operations are used for implementing debug capability at a source instruction level granularity. The operations are as follows:

- **PC Register Updates:** At the transition of source instructions code is generated to load the original PC address of the source instruction involved and store it in the PC register in the register file. In the main code no attempt is made to keep the PC register architecturally correct with respect to the original code as its value is implicit.
- **Instruction Predicate Operations:** These operations form a prefix to all source instructions and are used to allow instruction granularity execution control in the shadow code. They also serialize the operation groups associated with different host instructions so that they do not become intermixed.

In the shadow code all register file read and write operations are added into the CDFG. These are not optimised in any way so that the full register file state is maintained at source instruction boundaries. The shadow code is thus considerably larger than the main code. It is also much slower as many operations are serialised around the instruction predicate operations. However, this is not a major issue as the shadow code only needs to be executed when a breakpoint is activated and the code itself can be stored externally to code memory of a CriticalBlue processor.

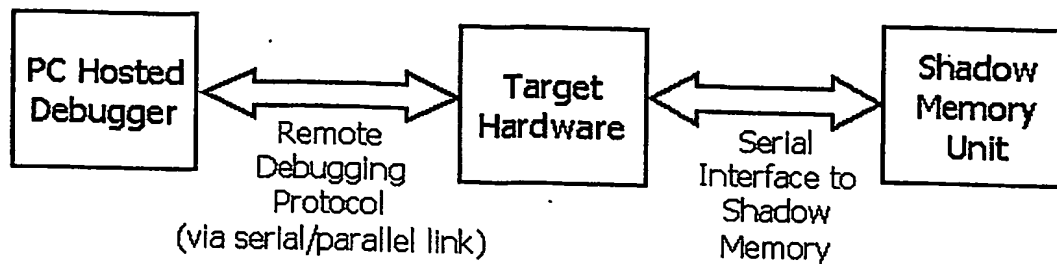
## **8.6 Debugging Translated Code**

Before an application is ever run on real hardware can be tested in the CriticalBlue simulation environment. This allows full cycle and bit accurate testing. Stimulus and behavioural modelling code will be produced to emulate the physical environment that the application will be executed within. This process will allow the detection of most major bugs in the application. The simulation runs natively using a C/C++ environment. The engineer is able to use his or her favourite debugger and integrated development environment.

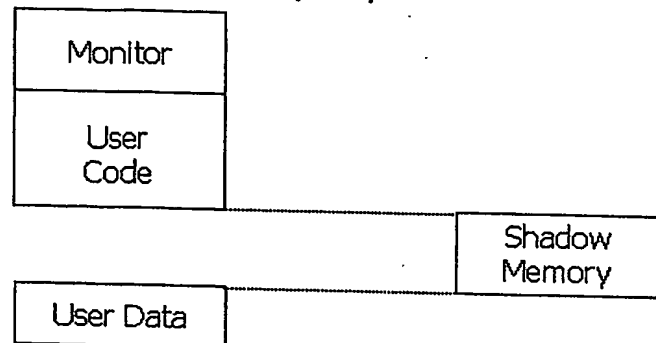
Of course, there are always likely to be application level bugs that only manifest themselves in the real hardware environment. To allow easy analysis of these, CriticalBlue supports a powerful debug environment. This is created from a combination of hardware structures embedded into every CriticalBlue processor, a monitor program present on the processor and special code produced by the CriticalBlue tools.

A CriticalBlue processor is able to provide a sufficiently accurate emulation of a 3<sup>rd</sup> party processor architecture that debug tools for that architecture can be used. The first release of the CriticalBlue tools will support the ARM architecture. ARM compatible symbolic debug tools running on a host PC platform can be used. This will provide source level debug capabilities.

The overall debug architecture is illustrated in the diagram below:



### System Memory Map



The host communicates with the target system via a serial or parallel link. A serial link may be used since high data speeds are not required and there is a need to minimise the area that the debug hardware occupies. A remote debugging protocol is run over the link. The host debugger can send commands to the target system to set breakpoints, read/write memory and read/write registers etc. When new commands are received by the target system they initiate interrupts which call handlers in the MetaMonitor code. These process the commands and send appropriate responses. The first version of the MetaMonitor will support the ARM Angel debug protocol. Thus a CriticalBlue system will be able to use any ARM debugger supporting the protocol.

A CriticalBlue processor also contains a number of breakpoint registers. These cause the processor to halt if the breakpoint is reached. They are utilised if breakpoints are set in the debugger. A breakpoint register consists of a region start address and a strand number. Execution is halted if the particular strand in the region is executed. This allows breakpoints that halt the machine on the equivalent of a particular original instruction in the code. All later strands are squashed so execution is effectively stopped at the source line, even if operations from later original instructions have been issued.

To allow high levels of parallelism in the architecture, code can be scheduled out-of-order with respect to the original code. Results may be generated in a completely different order to the way they are expressed in the original code. The user should not need to be aware of this. When they are debugging the code and single stepping through it they expect expressions to be evaluated and results produced in the sequential order expressed in the original code.

## 9 Translation Related Algorithms

### 9.1 Function Handling

#### 9.1.1 Function List Creation

```

FunctionList CreateFunctionList(UsedMap, FixupList)
// Examines all the code in the executable and generates a list of functions
// (in terms of address ranges) by examining destinations of calls and
// function pointer values. Ideally function entry points are associated
// with symbols so that they can be located again in the code database.
// However calls to addresses without a symbol are handled but the
// function has to be rescheduled on each occasion.
// UsedMap is returned as a bit map of all the locations in the executable
// image that are used and cannot be overwritten with translated code
// FixupList is a list of locations that need to be fixed up with the
// addresses of particular functions (for function return links and
// function pointers)

// we examine the executable image and create counts of entry point usage
// to try and identify individual functions - we also mark which words in
// the image cannot be modified by the translated code
UsedMap = cleared
FixupList = empty
entry_list = address 0 (main entry) and address 8 (SWI handler)
foreach word in the executable image do
    if word is function pointer data and not in filler code then
        // if we have a function pointer then we make sure the addressed
        // function is an entry point - we add the entry word to the fixup list
        // so that a link is written to the start of the function
        mark word as used in UsedMap
        add addressed function to entry_list with infinite usage count
        add addressed function to FixupList (so that link is written to start
        of function)
        if addressed function is marked as being implemented in hardware or
        is an interrupt function then
            report an error
        endif
    else if word is data and not in filler code then
        // all data is used except if it is in filler code (used for
        // shadow code) where the whole image can be overwritten
        mark word as used in UsedMap
    else if instruction is a SWI then
        // a SWI instruction needs to be presevered (for the immediate value)
        // and the following word is used to hold the link return address
        mark SWI instruction as used in UsedMap
        mark following instruction in FixupList with address of return from
        the SWI handler
    else if instruction is indirect branch for switch code then
        // for switch using an indirect branch we identify all the possible
        // destinations - each is written with a link to the translated
        // implementation of the code
        determine the range of possible values (by analysing range check code)
        foreach switch_dest in the possible range do
            mark switch_dest as used in UsedMap
            mark switch_dest in FixupList so that link to translated code is
            placed in the switch table (if the entry does an immediate direct
            branch then place a shortcut to the destination function)
            mark switch_dest with infinite usage count so that it is a separate
            entry point (although may become side entry of a region)
        endforeach
    else if instruction is a branch then
        // we increase the reference count of the destination to be able to
        // handle functions that are only entered using a branch

```

```

        add destination address to entry_list (or increase reference count
        if already in list)
    else if instruction is a branch and link (i.e. call) then
        // if we have a call then we ensure that the function becomes distinct
        // entry point and arrange for a link return address to be written
        add destination address to entry_list with infinite count
        mark following instruction in FixupList with address of return from
        the function
        if destination function is an interrupt function then
            // destination functions cannot be called directly as they have
            // additional code for preserving registers
            report an error
        endif
    endif
endif
endfor

// we now go through the potential function entry points
func_list = empty
foreach cand_func in entry_list do
    Initially make the function all code between two forced entry points (i.e.
    addressed functions or destinations of calls)
    Revisit all the code within the candidate function and reduce reference
    counts for branch destinations
    if there are non-zero reference counts in the function then
        Subdivide the function between each of the referenced addresses - this
        detects any other side entries that are branched to
    endif
    mark function as critical if entry symbol of function in critical list
    if function not on list of those implemented in hardware and
        function is not a code filler then
        // do not add behavioural model code for functions implemented in
        // hardware or filler code (to make executable big enough)
        CFLT = CreateCFLT(start address of function, end address of function,
        func_list (only backward information available))
        add function to func_list along with live inflow information from
        CFLT - this shows which register parameters are used by the function
        (non-volatile registers are masked off from the liveness even through
        they may appear live due to the save and restore code)
    endif
endfor

return func_list

```

## 9.2 Region Handling

### 9.2.1 Region List Creation

```

RegionList CreateRegionList(StartAddr, EndAddr, FuncType, FuncList)
// Identifies the regions within a particular function (as specified
// by an address range). Each region also has a loop depth that is used to
// weight requires for new resources during architectural synthesis.
// FuncType is the type of function being scheduled
// FuncList is the full list of functions in the code - this is used
// to generate better liveness information for the function

// get the control flow and liveness information for the function
CFLT = CreateCFLT(StartAddr, EndAddr, FuncList)

// determine the maximum number of original instructions in a block
if FuncType = Critical then
    // if we are creating a critical function then we do not limit the
    // number of instructions in a block
    MaxBlockIntrs = LARGE_VALUE

```

```

else
    MaxBlockInstrs = maximum instructions in a block - calculated as
                      a proportion of total function size (perhaps 25%) to
                      break up basic blocks for parallel scheduling in short functions
endif

// process the instructions in the function in ascending address order and
// create individual regions as required
region_list = empty
cur_addr = StartAddr
cur_depth = 0
do
    region_depth = cur_depth
    new_region = CreateRegion(cur_addr, EndAddr, cur_depth, FuncType,
                             MaxBlockInstrs, CFLT)
    add new_region to region_list with depth of region_depth
until cur_addr > EndAddr repeat

return region_list

```

## 9.2.2 Control Flow and Liveness Table Creation

```

CFLT CreateCFLT(StartAddr, EndAddr, FuncList)
// Generates the Control Flow and Liveness Table (CFLT) for the function
// with the given address range. The table contains information about
// control flow within the function including sources for back edges.
// Full liveness analysis is performed within the function by scanning
// instructions and determining which registers they use and define.
// If a function call can be found in the FuncList then the actual inflow
// parameters for the function can be used for the liveness - otherwise
// pessimistic assumptions must be made
// FuncList is the list of functions in the code along with their inflow
// liveness information (which may be slightly pessimistic if the
// function itself calls other functions)

CFLT = create edge information table with entry for inflows and outflows for
      each instruction in the range

// initially scan the instructions in the function to find the control flow
foreach cur_instr = StartAddr to EndAddr do
    if cur_instr is branch then
        if destination is in range of function then
            // update the CFLT with the appropriate edges
            create outflow edge information at cur_instr in CFLT
            create inflow edge information at destination instruction in CFLT -
            mark if the edge is backwards
        endif
        if cur_instr branch is conditional then
            // a pseudo edge is added for fall through cases
            add inflow to following instruction
        endif
    endif
endfor

// pass through the identified basic blocks and calculate the use and
// definition vectors of registers within the function
foreach basic_block in the CFLT do
    determine the use and definition masks for registers in the block - all
    calls and branches to named symbols in FuncList use the inflow
    liveness from the entry otherwise they are considered to use all
    possible parameter registers. All function calls are considered
    to define all volatile registers.
endfor

// now determine the in and out liveness of each basic block - this is
// done iteratively until a fixed point is reached

```

```

do
    foreach basic block in the CFLT do
        update the live in and live out of the block
    endfor
until no changes during pass repeat

return CFLT

```

### 9.2.3 Create Region

```

Region CreateRegion(StartAddr, EndAddr, CurDepth, FuncType,
                    MaxBlockInstrs, CFLT)
// Creates the CDFG for a region by translating original code between a
// given start and end address. If necessary unrolls loops by duplicating
// code to improve schedule potential.
// CurDepth is the current loop depth at the start of the region and is
// updated as required
// FuncType is the type of function being scheduled - if the function is
// critical then it should be for performance rather than code density
// MaxBlockInstrs is the maximum number of original instructions in a
// block - this may be limited to break up blocks in short functions
// to improve code density
// CFLT is the previously calculated Control Flow and Liveness Table (CFLT)

// reset variables for start of region
CDFG = empty
reg_defs = empty
is_entry_strand = true
num_dups = 1
total_weight = 0
total_instrs = 0
cur_addr = StartAddr
end_type = EndRegion
last_addr = maximum value

// If we are generating the main execution code then we need to generate a
// synchronisation branch to the start of the shadow code for the region.
// This branch is taken if a breakpoint is encountered.
if not generating shadow code then
    fixup_node = GenOp(CDFG, Wide Immediate Unit, shadow region start)
    mark fixup_node as being a fixup node
    (void) GenUnaryOp(CDFG, branch unit, synch branch, fixup_node)
endif

// build the region
do
    // Process the inflow edges for the current instruction. Avoid processing
    // an instruction for a second time (for instructions that cause a new
    // strand to be started). We also only process inflow edges on the
    // first unroll of a loop.
    if num_dups = 1 and cur_addr != last_addr then
        last_addr = cur_addr
        if there are inflow edges for cur_addr in CFLT then
            // if we have reached an inflow edge that has been branched
            // to from an earlier strand in the region then we can delete
            // the branch associated with the strand and reduce the number
            // of strands since the branch strand is gone
            foreach inflow visited inflow edge from CFLT do
                if branch was from strand in the current region then
                    // the branch is internal so delete the branch - the squash
                    // condition is inverted and its new destination is marked
                    // to squash all the strands after the branch and up to the
                    // destination
                    delete the branch node associated with the squashing_strand
                    if the squash strand has another fixed outflow then
                        // If the branch was originally unconditional then the squash

```

```

        // may be in a much earlier strand that has already been
        // fixed - we need to replicate the squash instruction. This
        // handles IF-THEN-ELSE code. The unconditional branch for the
        // end of the THEN clause modifies the original squash
        // instruction using an inverted condition to the ELSE
        // clause.
        replicate the squash instruction
    endif
    invert the squash condition
    mark the squash strand destination as the set of strands
    following the branching strand to the destination
    if the branch was the only operation in the strand then
        // The branch was in a seperate strand - it was conditonal
        // with a different condition to the previous strand. The
        // strand is no longer needed so the total number of
        // strands is reduced.
        delete the phase barrier associated with the strand
        delete the strand itself
    endif
endif
endif
endfor

    // keep an ongoing count of the loop depth
    CurDepth = CurDepth + number of back edges into cur_addr

endif
endif

end_type = EndRegion
if AdvanceToCode(CurAddr, EndAddr) then
    // find the end of the current block
    end_block_addr = cur_addr
    end_type = FindBlockEnd(CDFG, end_block_addr, MaxBlockInstrs, CFLT)

    // find out what the multiplier will be for the new strand
    strand_multiplier = CalcNewStrandMultiplier(CDFG, cur_addr, CFLT)

    // If this is an entry strand then the total weight of all previous
    // instructions in the region is halved. This lowers the overall
    // density and thus reduces the chance that the entry strand will be
    // accepted into the region. Multiple entry strands are inefficient
    // execution performance.
    if is_entry_strand then
        total_weight = total_weight / 2
    endif

    // increase the total instructions by the number in the block and
    // the weight by the number in the blocked weighted by the strand
    // multiplier
    total_instrs = total_instrs + (end_block_addr - cur_addr)
    total_weight = total_weight + strand_multiplier *
        (end_block_addr - cur_addr)

    // We calculate the execution density. This is the weighted number of
    // instructions divided by the total number of instructions. As more
    // conditional blocks are accepted into the region the execution
    // density is lowered.
    exec_density = total_weight / total_instrs

    // Get the minimum acceptable density. There is a different value for
    // critical functions where performance is emphasised over code
    // density.
    if FuncType = Critical then
        min_density = CRITICAL_MIN_DENSITY
    else
        min_density = NORMAL_MIN_DENSITY
    endif
endif

```

```

// Translate the block if adding it will keep the execution density at
// an acceptable level. This mechanism prevents too much conditional
// code being added to a region that will increase its length too much.
// If the conditional code is squashed then this significantly lowers
// the effective performance of the machine. If we run out of strands
// when translating the block then end the region.
if exec_density >= min_density then
    if not TranslateBlock(CDFG, cur_addr, end_block_addr, RegDefs,
        is_entry_strand, strand_multiplier, CFLT) then
        end_type = EndRegion
    endif
endif

// If the block ended due to a call site then update the total
// instructions appropriately. This lowers the execution density
// when a call is encountered. Calls are inefficient if the region
// has be returned to after the return. The inefficiency is related to
// the number of instructions so far (which have to be skipped over
// again on return to the same region). It is multiplied by the strand
// multiplier as if there is a smaller chance that the call will be
// executed anyway then there is less of an impact.
if end_type = CallSite then
    total_instrs = total_instrs + total_instrs * strand_multiplier
endif

// there is a limit on total instructions in the region
if total_instrs > MAX_INSTRUCTIONS then
    end_type = EndRegion
endif
endif

is_entry_strand = false
if end_type = InflowEdge then
    // The block ended due to an inflow edge. If all the edges are visited
    // then they are from previous strands and thus internal. Unvisited
    // edges cause the region to be ended for critical functions (which
    // emphasize performance over code density). Other regions may
    // continue but the next strand will be an entry strand and there
    // are limits on operations and strands.
    if there are any unvisited inflow edges for cur_addr in CFLT then
        is_entry_strand = true
        if FuncType = Critical or num_strands > MAX_CONTINUE_STRANDS or
            num_ops > MAX_CONTINUE_OPS then
            end_type = EndRegion
        endif
    endif
else if end_type = OutflowEdge then
    // The block ended due to an outflow edge. If there is no route
    // into the following code (i.e. the code is dead) then make sure
    // it is an entry strand. This also includes a return instruction.
    if outflow edge is unconditional and cur_addr has no inflow edges then
        is_entry_strand = true
    endif

    // reduce the loop depth as we pass backward outflow edges
    if if the outflow edge is backwards then
        CurDepth = CurDepth - 1
    endif

    // if we are scheduling a critical function and this looks like the
    // end of a loop (the outflow is backwards) then considering performing
    // code duplication to unroll the loop
    if FuncType = Critical and outflow edge is backwards then
        if edge dest is StartAddr and total_instrs < MAX_DUP_INSTRS and
            num_dups < MAX_DUPS and num_strands < MAX_DUP_STRANDS then
            // We duplicate code if we are branching back to the start of the
            // current region. There is a limit of the number of duplications,
            // the total number of instructions and the total strands thus far

```

```

    // (to avoid only part of an iteration being unrolled). We need
    // to modify the last branch in the loop and set the address
    // back to the start of the region.
    if outflow edge is unconditional then
        delete the branch as we are unrolling code to follow it
    else
        mark cur_strand as invert condition and change its destination
        to the current address (previous fall through)
    endif
    num_dups = num_dups + 1
    cur_addr = StartAddr
else
    // if we cannot unroll then we always end a region when a back
    // edge is detected in a critical function to avoid loaded
    // redundant code into a loop
    end_type = EndRegion
endif
endif
else if end_type = CallSite then
    // the next strand will be an entry strand to handle the return from
    // the call (this handles both direct and indirect calls)
    is_entry_strand = true
endif
until end_type = EndRegion repeat

// resolve the squashes by allocating strand numbers - any branches which
// are left in the code are to locations external to the region and are
// picked up as requiring fixups when generating the binary for the code
ResolveSquashes(CDFG, CFLT)

// If this is a region in a fast interrupt function then we generate a
// special return instruction in the final strand. This is used when
// locking down the code for the function during the initialisation
// sequence.
if FuncType = FastInterrupt then
    StartNewStrand(CDFG, true, FuncType)
    read_op = GenRegRead(CDFG, RegDefs, link register)
    GenBranch(CDFG, read_op)
endif

create new_region from CDFG
return new_region

```

## 9.2.4 Squash Dependency Update

```

void UpdateSquashDeps(CDFG, CurAddr, CFLT)
// Updates the squash dependency matrix. This details which strands precede
// the current one in the control flow. This is used for determining which
// predecessors are relevant when generating the register flow and other
// dependency information in the CDFG.
// CurAddr is the current instruction addressed being processed whose inflow
// edges are examined
// CFLT is the Control Flow and Liveness Table and is used to provide the
// required control flow edge information

deps_set = empty
foreach inflow edge to CurAddr from another strand in the CFLT do
    add the branching strand to the deps_set
    add the set of dependent strands for the branching strand itself to the
    deps_set
endfor
make the dependencies for the current strand equal to deps_set

```

## 9.2.5 Squash Code Resolution

```

void ResolveSquashes(CDFG, CFLT)
// Examines all the squashing strands and provides the final numbering for
// them. Generates the correct operands for the template squash instruction
// and generates additional squash operations for each strand as required.
// CFLT is the Control Flow and Liveness Table and shows the control flow.
// between strands

// number all the strands
strand_num = 0
foreach strand in the CDFG do
    allocate strand_num to strand
    if strand has not been resolved or has a conditional spur then
        // the strand perform an external branch (with a spur branch) or
        // is associated with a conditionally executed instruction with a
        // spur
        strand_num = strand_num + 1
    endif
    strand_num = strand_num + 1
endfor

foreach strand in the CDFG do

    // determine the set of strands that must be unconditionally squashed
    force_squash_set = empty
    if strand is an entry strand then
        // All entry strands (including the first strand) must squash all
        // strands that they cannot reach. Thus a strand containing
        // unreachable dead code will be squashed. Also if a call is in the
        // then part of an if-then-else then the entry strand after the
        // the call will squash the else part.
        for later_strand = strand + 1 to last strand do
            if later_strand cannot be reached from strand then
                add later_strand to force_squash_set
            endif
        endfor
    endif

    // determine the set of strands that must be conditionally squashed
    cond_squash_set = empty
    if dest is resolved (branch has been removed) then
        // if the strand is resolved (i.e. the associated branch instruction
        // is not still remaining) then it is to an internal strand so we
        // need to generate the correct squash conditions. If there are
        // multiple enclosed condition blocks then each must be set with
        // the same condition
        foreach later_strand = strand + 1 to last strand do
            if later_strand is dominated by strand and
               later_strand does not post-dominate all successors of strand then
                // the later strand can only be reached through the strand being
                // processed but it is not necessarily reached if the strand being
                // processed is reached. If the later strand is post-dominated by
                // the strand being processed then there is no requirement to set
                // a squash (one may have been set by an even earlier strand).
                add later_strand to cond_squash_set taking account of any invert
                condition on the strand
            endif
        endfor
    endif

    // set the actual squash values to be used in the CDFG
    modify squash instruction in to accommodate force_squash_set and
    cond_squash_set - add additional squash instructions if required

endfor

```

## 9.2.6 New Strand Multiplier Calculation

```

unsigned CalcNewStrandMultiplier(CDFG, BlockStartAddr, CFLT)
// Calculates the new strand multiplier for a new block. The strand
// multiplier is used to weight the number of instructions in the block
// in order to calculate the execution density. The execution density
// is then used to determine whether the block should be included in the
// current region or not.
// BlockStartAddr is the start address of the current block
// CFLT is the Control Flow and Liveness Table for the function

if there are no current strands then
    // the first strand in the region has a multiplier of 1
    strand_mult = 1
else if there are no inflows to the strand from the CFLT then
    // if the new strand has not been initiated because of a new basic block
    // (perhaps it was ended by a call or store) then we keep the same
    // strand multiplier as the previous strand
    strand_mult = strand multiplier from the previous strand
else if there is only one inflow to the strand from the CFLT and
    the previous strand ends with a conditional branch then
    // If there is only one inflow to a strand and it is via a conditional
    // branch. We make the multiplier half that of the previous strand on the
    // basis of a 50/50 chance of the branch being taken.
    strand_mult = half the strand multiplier from the inflow strand
else
    // If there are multiple inflows to the strand then we add the
    // multipliers of the preceeding strands together. Thus, for instance,
    // if this is the confluence point after a IF-THEN-ELSE construct then
    // the two clauses will have a multiplier of 0.5 - after confluence the
    // full multiplier of 1 is restored.
    strand_mult = 0
    foreach inflow to the strand that is internal using the CFLT do
        strand_mult = strand_mult + multiplier of predecessor strand
    endfor
endif

return strand_mult

```

## 9.3 Code Block Handling

### 9.3.1 Advancement To Code

```

bool AdvanceToCode(CurAddr, EndAddr)
// Advances the instruction pointer to the next block of executable code.
// Returns false if the end of the executable is reached.
// CurAddr is the current address that is advanced to point to code
// EndAddr is the end address of the executable

while CurAddr points to data and CurAddr < EndAddr do
    CurAddr = CurAddr + 1
endwhile

return CurAddr < EndAddr

```

### 9.3.2 Block End Determination

```

// possible reasons for ending a block translation
enum BlockEndType {
    EndRegion,    // the end of the region is reached

```

```

ExcessInstrs, // there is a limit of instructions in a block used to
              // break up long blocks in short functions into
              // different strands so they can be run in parallel for
              // achieving better code density
InflowEdge,  // if an inflow edge is detected then this ends
              // the basic block
OutflowEdge, // any branch or return instruction ends the block as
              // this delimits the basic block
StoreInstr,  // a store for which there are potentially aliased
              // loads in the same basic block - breaking them
              // into separate strands gives the opportunity to
              // speculative the load before the store
CallSite)    // a call (direct or indirect) ends a block as a new
              // strand is required after the return

EndType FindBlockEnd(CDFG, CurAddr, MaxBlockInstrs, CFLT)
// Finds the end of a block of instructions into the CDFG. A block continues
// until the basic block is ended, the excess instructions limit is reached,
// a call is reached or certain store instructions are reached. The extent of
// the block is determined prior to performing a translation of it. This
// allows a decision to be made whether the block should be included in the
// current region or a whether a new region should be started. Returns the
// reason for ending the block.
// CurAddr is the start address to search the block from - on return this
// points to the instruction after the end of the block
// RegDefs is a table of the current definers for all the architectural
// registers
// MaxBlockInstrs is the maximum number of original instructions that are
// permitted within a block - this may be limited to break up basic blocks
// in short functions to improve code density
// CFLT is the Control Flow and Liveness Table for the function

end_reason = EndRegion
intrs_in_block = 0
do
    // the instruction was successfully translated
    if instruction at CurAddr is a call (direct or indirect) then
        // we have encountered a call so we end the block as the following
        // code will be in a new strand
        end_reason = CallSite
    else if instruction at CurAddr writes to PC then
        // we have encountered a return
        end_reason = OutflowEdge
    else if there is an outflow edge for CurAddr in the CFLT then
        // we have encountered a branch - we mark it as being
        // visited so that the destination block can potentially
        // be included in the current region
        end_reason = OutflowEdge
        mark the destination of the edge as visited
    else if intrs_in_block >= MaxBlockInstrs then
        // there is a limit of instructions in a strand
        end_reason = ExcessInstrs
    else if there are any inflow edges for CurAddr in CFLT then
        // advance the address and see if there are any inflow edges to
        // the new address - the basic block is ended if so
        end_reason = InflowEdge
    else if the instruction at CurAddr is a store then
        // If the instruction is a store and there are load instructions
        // following it in the basic block that may be aliased (not
        // definitely or definitely not aliased) then the block is
        // ended. This allows the following load to be in a different
        // strand and thus potentially speculated earlier than the store
        // using chaz guards.
        foreach follow_instr in the same basic block do
            if follow_instr is a load that may be aliased to CurAddr then
                end_reason = StoreInstr
            endif
        endfor
    endfor
enddo

```

```

endif

// advance the address and instruction count
CurAddr = CurAddr + 1
instrs_in_block = instrs_in_block + 1

// continue until we have an end reason, we reach data or get to the end of
// of the executable
until end_reason != EndRegion or CurAddr >= EndAddr or
    CurAddr points to data repeat

// the block has been ended so return the reason why
return end_reason

```

### 9.3.3 Block Translation

```

bool TranslateBlock(CDFG, CurAddr, BlockEndAddr, RegDefs, IsEntryStrand,
    StrandMultiplier, CFLT)
// Translates a block of instructions into the CDFG. Translation continues
// until the block is ended or until the maximum number of strands limit is
// reached. The translation of the block will require the creation of one or
// more strands. Additional strands are required if conditional instructions
// are encountered. The extent of the block is predetermined by the
// FindBlockEnd function. Returns false if the maximum number of strands
// has been exceeded and the region must be ended.
// CurAddr is the first instruction in the block to be translated. This
// is updated by the routine.
// BlockEndAddr is the first instruction after the end of the block to be
// translated.
// RegDefs is a table of the current definers for all the architectural
// registers
// IsEntryStrand is true if the first strand in the block is an entry strand
// StrandMultiplier is the multiplier for calculating the execution weight
// of all strands in the block
// CFLT is the Control Flow and Liveness Table for the function

do
    // confluence the register definitions to take account of the inflow
    ConfluenceRegDefs(CDFG, CurAddr, RegDefs, CFLT)

    // a new strand is started - a guard operation and phase barrier
    // are constructed
    StartNewStrand(CDFG, IsEntryStrand, StrandMultiplier, FuncType)

    // update the squash dependency information
    if there are inflow edges at cur_addr then
        UpdateSquashDeps(CDFG, cur_addr, CFLT)
    else
        mark the new strand has having the same dependencies as the
        previous strand in addition to the previous strand itself
    endif

    // any further strands created for the block will not be entry strands
    IsEntryStrand = false

    // translate instructions until the end of the block or until a particular
    // instruction translation requests a new strand
    do
        continue_block = TranslateInstruction(CDFG, CurAddr, RegDefs, CFLT)
    until not continue_block or CurAddr = BlockEndAddr repeat

    // if there are any registers that have been defined in the strand and are
    // live outside it then they need a dependency on the sink node to ensure
    // the write occurs
    SinkLiveDefs(CDFG, CFLT, RegDefs, last instruction address)

```

```

// the register definition information is saved so that it may be
// retrieved by any subsequent confluence between strands
save the RegDefs information into the state for cur_strand

// End the region when the maximum number of strands have been used. A
// spare strand must always be left to handle a conditional strand for a
// conditional instruction. Another strand must be left spare for a
// region in a fast interrupt function as this has a special return
// strand. We do not perform the check if the instruction translation
// has requested a new strand as we cannot terminate the region at that
// point.
if continue_block then
    max_strands = MAX_STRANDS - 1
    if FuncType = FastInterrupt then
        max_strands = MAX_STRANDS - 2
    endif
    if current number of strands > max_strands then
        return false
    endif
endif
endif

// repeat until the end of the block unless we run out of strands
until CurAddr = BlockEndAddr repeat

// we successfully finished the block
return true

```

### 9.3.4 Live Register Sinking

```

void SinkLiveDefs(CDFG, CFLT, RegDefs, LastAddr)
// Sinks the live definitions at the end of a strand. Data dependency arcs
// are generated from each register definition in the current strand that
// is live at the end of the strand to a special sink node. This prevents
// the register write being optimised away.
// CFLT is the Control Flow and Liveness Table for the function
// RegDefs is the current set of register definitions
// LastAddr is the last instruction in the strand

live_set = the set of live out registers at LastAddr obtained from CFLT
foreach reg_def in RegDefs do
    if reg_def is in live_set and register was defined in current strand then
        create data flow arc from register definer to sink node for CDFG
    endif
endfor

```

### 9.3.5 Register Confluence Handling

```

void ConfluenceRegDefs(CDFG, CurAddr, RegDefs, CFLT)
// Updates the register definition information when inflow edges are
// reached. The register definitions are formed from the union of
// the register definitions flowing into all the incoming edges.
// Duplicate entries are removed but some registers might have a list
// of potential definers associated with them (if some register writes
// are in conditional sections of code)
// CurAddr is the instruction which has an inflow edge representing a
// new basic block
// RegDefs is the table of current definers for the architectural
// registers - this is updated to represent the inflow for the new
// basic block
// CFLT is the Control Flow and Liveness Table that is also used to
// hold the register definers at the end of previous basic blocks

delete the current RegDefs

```

```

clear the first use for all registers
foreach inflow edge from another strand in the region do
    add the register definitions left at the end of the strand into the
    RegDefs - removing duplicate entries
endfor

```

### 9.3.6 New Strand Creation

```

void StartNewStrand(CDFG, IsEntryStrand, StrandMultiplier, FuncType)
// Initiates a new strand. A guard operation and a phase barrier are
// generated for the new strand. Dependencies are generated onto
// any preceeding squash operation for the strand, any preceeding phase
// barriers and any preceeding branch.
// IsEntryStrand is true if the strand is an entry point to the region
// StrandMultiplier is a multiplier for the strand for calculating the
// execution weight. Conditional strands are given a lower multiplier on
// the assumption that each conditional branch has a 50% chance of being
// taken.
// FuncType is the type of function being scheduled

create new strand entry in the CDFG
mark the new strand information with the IsEntryStrand flag
mark the new strand with the StrandMultiplier

// Each strand has a guard operation that is dependent on the phase
// barrier (i.e. it must be issued beforehand). This is a conditional node
// that is only issued if there is a weak dependency arc violation. It
// aborts the strand if it is not the lowest numbered strand being executed.
guard_op = GenOp(CDFG, guard unit, guard)
if IsEntryStrand and entry is for true side entry (not call return) then
    mark guard_op so conditional arcs are not connected to it - it must always
    be issued as this is an entry strand
endif

// Commit nodes separate speculative and committed phases. Commit
// nodes ordered themselves (so order of strands maintained). Non
// speculative instructions such as stores and register writes behind
// the commit.
phase_op = GenOp(CDFG, commit unit, commit)

// the guard must come before the committed phase of the strand
generate control arc between guard_op and phase_op

if there is a current squash instruction then
    if FuncType = Critical then
        // A strong arc is generated between any squash instruction associated
        // with the strand and the phase barrier. This ensures that the
        // committed phase of the strand only starts when its squash status
        // is known.
        create a strong control arc from preceeding squash instruction to
        phase_op
    else
        // A weak arc is generated between any squash instruction associated
        // with the strand and the phase barrier node itself. The conditional
        // arc corresponding to the weak arcs is to the guard operation
        // for the strand. Thus if the squashes for the strand are not resolved
        // by its committed phase then the strand is aborted unless it is the
        // lowest strand being executed.
        create a weak control flow arc from the preceeding squash instruction
        to phase_op
        create a corresponding conditional arc between the squash instruction
        and guard_op
    endif
endif

if there is a preceeding strand then

```

```

if FuncType = Critical then
    // A strong control arc is generated to the phase barrier of any
    // previous strand - this enforces the strand commit order.
    create a strong control flow arc from the phase barrier to the
    preceeding strand phase_op
else
    // A weak control arc is generated to the phase barrier of any
    // previous strand - this enforces the strand commit order. The
    // conditional arc is to the current strand guard so that if the
    // strands are committed out of order then the guard operation is
    // activated.
    create a weak control flow arc from the phase barrier of the preceeding
    strand to phase_op
    create a corresponding conditional arc between the preceeding phase
    barrier and guard_op
endif

if the previous strand has a branch operation then
    if FuncType = Critical then
        // If the previous strand has a branch operation then a strong
        // control arc is generated to it. This ensures that the branch is
        // issued before the current strand commits so that the current
        // strand may be aborted if required.
        create a strong strong control flow arc from the previous branch to
        phase_op
    else
        // If the previous strand has a branch operation then a weak control
        // arc is generated to it - this ensures that the branch is issued
        // before the current strand commits so that the current strand may
        // be aborted if required. The conditional as to the current strand
        // guard.
        create a weak control flow arc from the previous branch to phase_op
        create a corresponding conditional arc from the previous branch to
        guard_op
    endif
endif
endif
endif

```

## 9.4 Code Translation

### 9.4.1 Instruction Translation

```

bool TranslateInstruction(CDFG, CurAddr, RegDefs, CFLT, FuncType)
// Translates the given source instruction into entries within the CDFG.
// The instruction is transformed into primitives to read operand registers,
// perform the operation on the required execution unit and then write back
// the modified registers. Some operations (such as load/store multiple on
// ARM) may be transformed into many operations. Instructions
// with conditions, such as conditional branches and conditional
// instructions, are broken down into the conditional evaluation into a
// squash instruction, followed by the actual instruction translation in
// the following strand. In such a case the function requests a new strand
// without updating the current address.
// Returns false if a new strand is required. This may be the case if
// a call, branch or software interrupt is translated. A new strand may
// also be required when translating conditional instructions and the
// condition for the current strand is not that required.
// CDFG is the current Control and Data Flow Graph that is updated as
// required
// CurAddr is the address of the instruction that should be translated -
// this is updated by this routine
// RegDefs is a table of the current definers for all the architectural
// registers - this is updated as required and multiple definers
// might be created if a register update is conditional
// CFLT is the Control Flow and Liveness Table for the region
// FuncType is the type of function being scheduled

```

```

// at the start of a new strand we know that the entry conditions will be
// that of the condition of the instruction
if no operations have been issued in the current strand then
    make the current strand condition the same as that of the current
    instruction
endif

// Generate an instruction PC load and predicate. We do not generate a
// second predicate for the same instruction. If an instruction is
// conditional then the predicate will be generated before the condition
// evaluation and squash. The code for the instruction will actually be in
// the following strand. Thus any breakpoint will occur whether or not the
// condition is true. However, if there are subsequent instructions with the
// same condition then they will be in the conditional strand. Thus the
// breakpoint check will only occur conditionally. This differs from the
// normal breakpoint behaviour. We also generate any special entry code for
// the function entry.
if last_pred != CurAddr then
    updated_PC = GenPCUpdate(CDFG, CurAddr)
    GenInstrPredicate(CDFG, updated_PC)
    last_pred = CurAddr

    // generate any required entry code if this is the first instruction
    // in a function - the code is generated after the instruction predicate
    // for the first instruction so effectively executes as part of its
    // behaviour
    if this is the first instruction in the function then
        GenEntryCode(CDFG, RegDefs, CurAddr, FuncType)
    endif
endif

// translate the current instruction
if instruction condition is not the same as that of the strand then
    // If we have a conditional instruction and the condition for the strand
    // is not the same then we evaluate the condition - which will generate a
    // squash instruction. We then end the current strand without advancing
    // the instruction address. When we try and translate the same instruction
    // again we will have set the correct condition for the strand.
    GenCondEval(CDFG, condition from instruction)
    return false
else
    // the strand condition is correct so translate the instruction
    if instruction is branch then
        // We translate a branch and move onto the next instruction - however
        // the current strand is ended. If the branch was conditional then the
        // branch will have been produced in the conditional strand.
        GenBranch(CDFG, destination address)
        CurAddr = CurAddr + 1
        return false
    else if instruction is branch and link then
        if instruction is branch to hardware implemented function then
            // we have a software call now implemented as a functional unit - we
            // translate the call but we do not need to end the strand
            TranslateHardwareFunc(CDFG, RegDefs, CurAddr, FuncType)
        else
            // We translate the call and move onto the next instruction - however
            // the current strand is ended. If the call was conditional then the
            // code will have been produced in the conditional strand.
            TranslateCall(CDFG, RegDefs, CurAddr, FuncType)
            CurAddr = CurAddr + 1
            return false
        endif
    else if instruction is software interrupt then
        // software interrupts are translated like calls and they end the
        // strand
        TranslateSoftwareInt(CDFG, RegDefs, CurAddr, FuncType)
    endif
endif

```

```

    CurAddr = CurAddr + 1
    return false
else if instruction is data processing then
    TranslateDataProc(CDFG, RegDefs, CurAddr, FuncType)
else if instruction is multiply then
    TranslateMultiply(CDFG, RegDefs, CurAddr, FuncType)
else if instruction is memory access then
    TranslateAccessSingle(CDFG, RegDefs, CurAddr, FuncType)
else if instruction is multiple memory access then
    TranslateAccessMultiple(CDFG, RegDefs, CurAddr, FuncType)
else if instruction is swap then
    TranslateSwap(CDFG, RegDefs, CurAddr, FuncType)
else if instruction is CPSR access then
    TranslateCPSRAccess(CDFG, RegDefs, CurAddr, FuncType)
else
    instruction type is unknown so report an error
endif

// we advance to the next instruction and indicate that the instruction
// was translated successfully
CurAddr = CurAddr + 1
return true
endif

```

## 9.4.2 Entry Code Generation

```

void GenEntryCode(CDFG, RegDefs, CurAddr, FuncType)
// Generates special entry code for a function. For the main entry point to
// the program this generates code to setup the interrupt handlers. For
// interrupt handlers themselves this adds the extra code to preserve the
// volatile registers. For the SWI handler this adds extra code to store
// the required registers.
// RegDefs is the current register definitions
// CurAddr is the address of the first instruction in the function
// FuncType is the type of function being scheduled

// if the function is the main entry point then we need to setup the
// system interrupts
if CurAddr = the main entry point then
    // We setup all the fast interrupts. The handler functions are loaded into
    // the instruction buffer and then locked down so that they are always
    // available. This is achieved by calling the final strand in a special
    // mode. This locks down the entry in the instruction buffer and
    // immediately returns. Unlike with normal calls the actual return address
    // is loaded into the link register. This actually requires multiple
    // strands so needs to return multiple times for a new strand.
    generate code to clear instruction buffer and reset base register to 0 and
    disable interrupts
    foreach fast interrupt function do
        // set up ready for the call
        immed_op = GenImmediate(CDFG, direct return value, true)
        GenRegWrite(CDFG, RegDefs, link register, immed_op, main result port,
                    FuncType)
        entry = the address of the fast interrupt entry from the configuration
                file (fixed up address) with the last strand which
                immediately returns

        // perform a branch but use a special method for calling so that it
        // causes a lock down
        GenBranch(CDFG, entry)
    endfor

    // we setup the the entry points for the interrupts into the appropriate
    // branch registers
    foreach interrupt (fast and slow) do
        entry = the address of the slow interrupt from the configuration file
    endfor

```

```

                                (fixed up address)
        generate a method on the branch unit to load the entry point into the
        interrupt branch area
    endfor
    enable interrupts

else if FuncType = SlowInterrupt or FuncType = FastInterrupt then
    // If the function is an entry to an interrupt then we have to generate
    // special code to save the contents of all the volatile registers - these
    // are not normally saved as part of the entry sequence. Special registers
    // are available for storing the values - nested interrupts are not
    // permitted so they are not overwritten.
    foreach volatile register, stack pointer and condition code registers do
        generate code to load volatile register then store into special storage
        register
    endfor

    // interrupt routines have their own private stack (address is
    // defined in the configuration file) - this is required because the
    // state of the stack cannot be guaranteed at a cycle boundary level in
    // the main code
    generate code to load stack pointer with fixed value from MetaMapper
    command line option
    endif
else if FuncType = SWIHandler then
    // If the function is the entry to the SWI handler then we save copies of
    // various registers into other registers. This emulates the behaviour
    // of a real ARM processor that enters SVC mode when entering the SWI
    // handler. The stack pointer and link registers are saved along with
    // the condition codes.
    save a copy of the stack pointer to the r13_svc register
    save a copy of the link register to the r14_svc register
    save a copy of the condition code registers into extra registers
    copy the alternative link register to the main link register
endif

```

### 9.4.3 Call Translation

```

void TranslateCall(CDFG, RegDefs, CurAddr, FuncType)
// Translates a call operation. This is a direct call operation -
// indirect calls are handled elsewhere as writes to the PC register
// with a separate explicit write to the link register. The return
// address is loaded to the link register as an immediate value. The
// load is marked as a fixup node to avoid the function being
// rescheduled for each PC address change. The branch is then
// performed in the same strand (if it is a conditional branch then
// a special strand will have already been created). All architectural
// registers are cleared as subsequent code does not need a dependency
// as the call return restarts the strand.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of function being scheduled

// the link register is written at the end of the strand prior to
// the actual branch being performed
immed_op = GenImmediate(CDFG, PC value after BL instruction, true)
mark immed_op as a fixup node
GenRegWrite(CDFG, RegDefs, link register, immed_op, main result port,
            FuncType)

// the actual branch is generated to be dependent on the phase barrier for
// the current strand - it must be issued before the barrier
GenBranch(CDFG, destination PC value)

// Any current register definitions are cleared - this prevents any register
// dependencies being generated to before the call. Although non-volatile

```

```
// registers maintain the same value this is not relevant to schedule
// dependencies since the region is restarted after the call. Even return
// registers are not made dependent because the region is restarted.
foreach architectural register do
    clear any definitions for the register
endfor
```

## 9.4.4 Hardware Function Translation

```
void TranslateHardwareFunc(CDFG, RegDefs, CurAddr, FuncType)
// Translates a hardware function call. This is a software function call in
// the original code to a function that is noted as being implemented in
// hardware. The input parameters are passed as operands to the hardware
// unit. Multiple parameter returns are supported by writing to addresses
// pointed to by reference parameters.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of function being scheduled

// obtain the details for the hardware operation to be performed
hardware_unit = the unit to be used as obtained from the configuration file
hardware_method = the method to be used as obtained from the configuration
                  file

// Load the required parameters to be made available as operands to
// the hardware operation. All the input parameters are loaded - from
// registers or stack as required.
in_params = empty
foreach param_num parameter to function do
    if param_num is an input parameter and
        not the this parameter for a C++ function then
        if parameter is in a register then
            // simply read a register parameter
            param_op = GenRegRead(CDFG, RegDefs, param_num register, FuncType)
        else
            // a later parameter must be read from the appropriate location on
            // the stack frame
            immed_op = GenImmediate(CDFG, stack frame offset of parameter, false)
            base_op = GenRegRead(CDFG, RegDefs, stack pointer)
            addr_op = GenBinaryOp(CDFG, addition unit, add, base_op, immed_op)
            param_op = GenUnaryOp(CDFG, memory unit, load, addr_op)
            AddMemoryDependencies(CDFG, param_op, FuncType)
        endif
        // the parameter is added to those passed to the hardware unit
        add param_op (main result port) to in_params
    endif
endfor

// perform the actual hardware operation
hw_op = GenOp(CDFG, hardware_unit, hardware_method, in_params)

// if the operation produces a result then write to the result
// architectural register
if the function has a return parameter then
    GenRegWrite(CDFG, RegDefs, register 0, hw_op, main result port,
                FuncType)
endif

// Store any output parameters from the hardware operation. These are
// reference parameters. The input parameters are loaded to obtain the
// address and then the result port from the hardware unit is read and the
// data written to the appropriate location. Note that all such results must
// be word sized.
foreach param_num parameter to function do
    if param_num is an output parameter then
```

```

// obtain the parameter address as required from either a register
// or from the appropriate stack frame parameter location
if parameter is in a register then
    addr_op = GenRegRead(CDFG, RegDefs, param_num register, FuncType)
else
    immmed_op = GenImmediate(CDFG, stack frame offset of parameter, false)
    base_op = GenRegRead(CDFG, RegDefs, stack pointer, FuncType)
    addr_op = GenBinaryOp(CDFG, addition unit, add, base_op, immmed_op)
    addr_op = GenUnaryOp(CDFG, memory unit, load, addr_op)
    AddMemoryDependencies(CDFG, addr_op, FuncType)
endif

// write the result to the memory location
write_op = GenBinaryOp(CDFG, memory unit, store, addr_op, hw_op
    (appropriate result port))
AddMemoryDependencies(CDFG, write_op, FuncType)
endif
endif

```

### 9.4.5 Software Interrupt Translation

```

void TranslateSoftwareInt(CDFG, RegDefs, CurAddr, FuncType)
// Translates a software interrupt. This is converted into a call
// to the entry point of the SWI handler at address 8. The return
// address is loaded into a special link register. This is copied
// into the main link register by special entry code for the SWI
// handler. This emulates the behaviour of a real ARM processor
// that switches execution modes to provide extra copies of some
// registers. The SWI instruction is preserved to allow the handler to
// read the 24 bit immediate field from the instruction. The following
// instruction is used for a link return.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of function being scheduled

// write the return address to a special alternative link register -
// this is read by the SWI call handler and then placed into the link
// register after initially preserving the old link register
immmed_op = GenImmediate(CDFG, PC value after SWI instruction, true)
mark immmed_op as a fixup node
GenRegWrite(CDFG, RegDefs, alternative link register, immmed_op,
    main result port, FuncType)

// perform the actual branch to the SWI handler
GenBranch(CDFG, SWI call handler)

```

### 9.4.6 Data Processing Instruction Translation

```

void TranslateDataProc(CDFG, RegDefs, CurAddr, FuncType)
// Translates all the data processing instructions. These are binary
// operations that mostly write back to a destination register. All
// addressing modes are handled and all the necessary flags generated.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of function being scheduled

unit_type = type of unit allocated for instruction
method = the method for the instruction
if addressing mode is immediate then
    if instruction is add and S is not set and Rn is PC and
        addressing mode is immediate then
        // if the instruction adds an immediate offset to the PC register then
        // this is converted into a direct latch of an immediate address - the

```

```

    // operation is marked as a fixup so any change of address does not
    // force re-scheduling of the function
    addr = calculated immediate address using PC + immediate offset
    if addr is not in a data area then
        generate an error
    endif
    op = GenImmediate(CDFG, addr, true)
    mark op as being a fixup instruction
else
    // if the operation uses a register with an immediate second
    // operand then make the two operands available and then perform the
    // required operation
    op1 = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
    op2 = GenImmediate(CDFG, immediate value, false)
    op = GenBinaryOp(unit_type, method, op1, op2)
endif
else if addressing mode is immediate shift register then
    // the operation shifts the second parameter by a fixed amount -
    // form the required second operand, then the first operand and then
    // perform the required function
    if shift amount is not 0 then
        // form the required second operand - it is formed first to avoid
        // disturbing the first operand
        op2 = GenShiftImmed(CDFG, RegDefs, shift type from Sh field, op2, shift
            amount, instruction is not arithmetic, FuncType)
    else
        // if the shift amount is 0 then this is a register-register operation
        op2 = GenRegRead(CDFG, RegDefs, Rm operand, FuncType)
    endif
    op1 = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
    op = GenBinaryOp(CDFG, unit_type, method, op1, op2)
else
    // addressing mode must be register shift with a register amount - the
    // second operand is formed, then the first operand is formed and then
    // the operation is performed
    op2 = GenShiftReg(CDFG, RegDefs, shift type from Sh field,
        Rm operand, Rs operand, instruction is not arithmetic, FuncType)
    op1 = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
    op = GenBinaryOp(CDFG, unit_type, method, op1, op2)
endif

// all but the test and compare operations write back the result to an
// architectural register
if instruction is not Test or Compare (that do not write
    write to destination register) then
    GenRegWrite(CDFG, RegDefs, Rd operand, op, main result port, FuncType)
endif

// if the S flag is set (and we are not writing to the PC register) then
// the condition codes need to be set - the values are obtained by reading
// the extra output ports from the unit performing the operation
if S flag set (for writing condition codes) and
    destination register is not PC then
    // the N and Z flags are set for all operations
    GenRegWrite(CDFG, RegDefs, register for N flag, op, N result port,
        FuncType)
    GenRegWrite(CDFG, RegDefs, register for Z flag, op, Z result port,
        FuncType)
    if operation was arithmetic then
        // the C and V flags are only set for arithmetic operations
        GenRegWrite(CDFG, RegDefs, register for C flag, op, C result port,
            FuncType)
        GenRegWrite(CDFG, RegDefs, register for V flag, op, V result port,
            FuncType)
    endif
endif
endif

```

### 9.4.7 Multiply Translation

```

void TranslateMultiply(CDFG, RegDefs, CurAddr, FuncType)
// Translates a multiply instruction. Both 32 bit and 64 bit variants of the
// instructions are supported. The multiply-accumulate forms are supported
// using the standard adder for the accumulate. The implementation relies on
// a 32 bit by 32 bit multiplier producing a 64 bit result.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of function being scheduled

// perform the basic multiply operation - this multiplies two 32 bit
// values to produce a 64 bit result
left_op = GenRegRead(CDFG, RegDefs, Rm operand, FuncType)
right_op = GenRegRead(CDFG, RegDefs, Rs operand, FuncType)
mul_op = GenBinaryOp(CDFG, multiply unit, signed/unsigned multiply depending
                    on instruction, left_op, right_op)

// handle the remainder of the instruction depending on its type
if instruction is MUL then
    // a simple multiply writes back a 32 bit result
    GenRegWrite(CDFG, RegDefs, Rd operand, mul_op, lower result port,
                FuncType)
else if instruction is MLA then
    // if the operation is a 32 bit multiply-accumulate then read the
    // accumulator register, add the result and then write back the
    // accumulator
    acc_op = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
    add_op = GenBinaryOp(CDFG, addition unit, addition, mul_op, acc_op)
    GenRegWrite(CDFG, RegDefs, Rd operand, add_op, main result port,
                FuncType)
else if instruction is UMULL or SMULL then
    // if the operation is a 64 bit signed or unsigned multiply then
    // write back the full 64 bit result into two registers
    GenRegWrite(CDFG, RegDefs, RdLo operand, mul_op, lower result port,
                FuncType)
    GenRegWrite(CDFG, RegDefs, RdHi operand, mul_op, upper result port,
                FuncType)
else
    // if we have a 64 bit multiply accumulate then we must read and then
    // add to the 64 bit accumulator - an add with carry is used to form
    // the upper 32 bits of the result
    acc_op = GenRegRead(CDFG, RegDefs, RdLo operand, FuncType)
    add_op = GenBinaryOp(CDFG, addition unit, addition, mul_op -
                        lower result port, acc_op)
    GenRegWrite(CDFG, RegDefs, RdLo operand, add_op, main result port,
                FuncType)
    acc_op = GenRegRead(CDFG, RegDefs, RdHi operand, FuncType)
    param_list = mul_op - upper result port
    add acc_op to param_list
    add add_op carry output port to param_list
    add_op = GenOp(CDFG, addition unit, addition with carry, param_list)
    GenRegWrite(CDFG, RegDefs, RdHi operand, add_op, main result port,
                FuncType)
endif

// if the S bit of the instruction is set then the N and Z flags need to
// be updated - N is the most significant bit of the result and Z is set
// if the whole result is 0
if S bit is set then
    bit copy from bit 31 of upper result word to N flag
    set Z flag if result (one or two words) is zero - use OR operation
    for two word case
endif

```

### 9.4.8 Single Memory Access Translation

```

void TranslateAccessSingle(CDFG, RegDefs, CurAddr, FuncType)
// Translates a single memory access instruction. This might be for
// a word, half word or byte. All addressing modes are handled along
// with write backs to the base register for pre/post increment modes.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of function being scheduled

// calculate the address to use for the load
if address is pre-indexed then
    // calculate the full address if pre-indexing - and write it back
    // to the base register if required
    addr_op = GenerateAddress(CDFG, RegDefs, CurAddr, FuncType)
    if write back bit is set then
        // write back the updated base register
        GenRegWrite(CDFG, RegDefs, base register Rn, addr_op, main result port,
                    FuncType)
    endif
else
    // the address is post-indexed so just use the base register for the
    // address
    addr_op = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
endif

// perform the actual memory access - adding more dependencies as required
// to maintain semantically correct memory operation order
if instruction is a store then
    // read the source register for a store - shift to the correct position
    // for a subword write
    reg_op = GenRegRead(CDFG, RegDefs, Rd operand, FuncType)
    if store is for byte or half word then
        reg_op = GenerateOperation(shifter unit, byte shift operation,
                                   reg_op, addr_op)
    endif
    op = GenBinaryOp(CDFG, memory unit, store method, addr_op, reg_op)
else
    // For a load read the location and then shift to the correct byte (and
    // sign extend if required) if a subword access. Write the value back to
    // the required architectural register.
    op = GenUnaryOp(CDFG, memory unit, load method, addr_op)
    if load is for a byte or half word then
        op = GenBinaryOp(CDFG, byte shifter, byte shift and sign extend,
                         op, addr_op)
    endif
    GenRegWrite(CDFG, RegDefs, Rd operand, op, main result port, FuncType)
endif
AddMemoryDependencies(CDFG, op, FuncType)

// if the address is post indexed then generate the new address and write
// it back to the base register (it is always written back in the
// post-index case)
if address is post-indexed then
    addr_op = GenerateAddress(CDFG, RegDefs, CurAddr, FuncType)
    GenRegWrite(CDFG, RegDefs, base register Rn, addr_op, main result port,
                FuncType)
endif

```

### 9.4.9 Multiple Memory Access Translation

```

void TranslateAccessMultiple(CDFG, RegDefs, CurAddr, FuncType)
// Translates a multiple load or store instruction. This allows any number
// of the architectural registers to be stored or loaded into consecutive
// memory locations. This is generally used to preserve non-volatile

```

```

// registers in a function. However, the instruction may also be used to
// perform fast block moves. It is often the case that the load multiple
// might restore the PC register - in which case it also effects a return.
// The PC register is always the last to be loaded as it is the highest
// numbered register. The register write operation produces code to handle
// the return if the PC register is written to.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of the function being scheduled

if register list is not empty then
    // Read the base register - offsets are generated from this for the
    // individual access. Seperate offsets are used to allow the accesses to
    // be reordered.
    base_op = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)

    // load or store each of the registers in the list
    offset = 0
    for reg_num = 0 to 15 do
        if reg_num is in register list then
            // generate the next immediate offset
            if U bit is set (increment address) then
                new_offset = offset + 4
            else
                new_offset = offset - 4
            endif

            // if pre-indexing is being used then update the offset
            if P bit set (pre-index) then
                offset = new_offset
            endif

            // generate the required address - if the offset is 0 then it is just
            // the base address otherwise an addition is used to generate it
            if offset != 0 then
                immed_op = GenImmediate(CDFG, offset, false)
                addr_op = GenBinaryOp(CDFG, addition unit, add method, base_op,
                                     immed_op)
            else
                addr_op = base_op
            endif

            // perform the actual individual load or store operation
            if L bit is set (a load multiple is being performed) then
                mem_op = GenUnaryOp(CDFG, memory unit, load, addr_op)
                GenRegWrite(CDFG, RegDefs, reg_num, mem_op, main result port,
                           FuncType)
            else
                data_op = GenRegRead(CDFG, RegDefs, reg_num, FuncType)
                mem_op = GenBinaryOp(CDFG, memory unit, store, addr_op, data_op)
            endif

            // generate any required memory dependencies to maintain semantically
            // correct operation order
            AddMemoryDependencies(CDFG, mem_op, FuncType)

            // the offset is updated
            offset = new_offset
        endif
    endfor

    // update the value of the base register if required
    if W bit is set (write back base register) then
        immed_op = GenImmediate(CDFG, offset, false)
        addr_op = GenBinaryOp(CDFG, addition unit, add method, base_op,
                             immed_op)
        GenRegWrite(CDFG, RegDefs, Rn operand, addr_op, main result port,
                   FuncType)
    end

```

```
endif
endif
```

### 9.4.10 Swap Translation

```
void TranslateSwap(CDFG, RegDefs, CurAddr, FuncType)
// Translates swap instructions. These are atomic memory operations used
// to swap two values in memory in order to implement semaphore operations.
// The load and the store must not be separated. Since this is encoded as a
// single instruction it is guaranteed that the CriticalBlue operations will
// be
// part of the same strand - and thus remain atomic.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of the function being scheduled

// load the present value in the memory location - if it is a
// byte value then the byte shifter unit is used to extract the individual
// byte
base_addr = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
load_op = GenUnaryOp(CDFG, memory unit, load, base_addr)
AddMemoryDependencies(CDFG, load_op, FuncType)
if load is for a byte or half word then
    load_op = GenBinaryOp(CDFG, byte shifter, byte shift and sign extend,
                          load_op, base_addr)
endif

// Write the new value to the location - if it is a byte value
// then the byte shifter is used to place the byte in the correct
// part of the word prior to the write. The value read by the load
// is written back to the register - after the register read to
// preserve the semantics if the same register is used for both.
st_data = GenRegRead(CDFG, RegDefs, Rm operand, FuncType)
if swap is for a byte then
    st_data = GenBinaryOp(CDFG, byte shifter, byte shift operation,
                          st_data, base_addr)
endif
GenRegWrite(RegDefs, RegDefs, Rd operand, load_op, main result port,
            FuncType)
store_op = GenBinaryOp(CDFG, memory unit, store, base_addr, st_data)
AddMemoryDependencies(CDFG, store_op, FuncType)
```

### 9.4.11 Status Word Access Translation

```
void TranslateCPSRAccess(CDFG, RegDefs, CurAddr, FuncType)
// Translates Current Program Status Register (CPSR) accesses. Only the
// condition code flag bits are modelled in the CriticalBlue architecture -
// all other bits are ignored. Each of the flag bits is stored in an
// individual flag register in the CriticalBlue architecture. Code is
// emitted
// to convert this representation to/from the standard bit positions in
// the CPSR.
// RegDefs is the current register definitions
// CurAddr is the address of the CPSR access instruction
// FuncType is the type of the function being scheduled

if operation is CPSR read then
    // The flag bits are actually stored in individual registers - this
    // combines them into a single value equivalent to what would be read
    // from a real CPSR register. A basic set of flags (for other status
    // information) is combined with the results of reads for each of the
    // required flag registers
    base_op = GenImmediate(CDFG, default value for CPSR - all flags reset,
                          true)
```

```

foreach of the flag registers N, Z, C and V do
  read the appropriate register
  use bit test and set to copy the bit to the appropriate part of base_op
endfor
GenRegWrite(CDFG, RegDefs, Rd operand, last update operation,
            main result port, FuncType)
else if immediate write then
  // A fixed set of flag values are being written from an immediate field -
  // all but the flag bits are ignored. Set up instructions to write 0s or
  // 1s to the flag registers as appropriate.
  flag_imm = the immediate value to be loaded into CPSR
  foreach of the flag registers N, Z, C and V do
    set/reset the status of the individual flag registers as required by
    flag_imm (no flag bits ignored)
  endfor
else
  // The CPSR is being set from a register. The source register is read and
  // then the individual flag bits are copied into the appropriate flag
  // registers. Other non-condition code bits are ignored.
  reg_op = GenRegRead(CDFG, RegDefs, Rm operand, FuncType)
  foreach of the flag registers N, Z, C and V do
    read the appropriate flag bit from reg_op
    set/reset the status of the individual flag register
  endfor
endif
endif

```

#### 9.4.12 Address Generation

```

Node GenerateAddress(CDFG, RegDefs, InstrAddr, FuncType)
// Translates the address calculation for a memory access. Handles both
// standard and half-word/signed byte forms of memory access. Accesses to
// data from PC offsets are translated into immediate loads of the required
// location. Returns the address generating operation.
// RegDefs is the current register definitions
// InstrAddr is the instruction being translated
// FuncType is the type of the function being scheduled

if instruction offset is immediate then
  if base register Rn is PC then
    // a PC base register plus an immediate offset is transformed into an
    // immediate address load - marked as a fixup to handle differences in
    // the address without forcing the re-scheduling of code
    addr = calculated immediate address using PC and immediate offset value
           (handling split immediate field if required)
    if addr is not in a data area then
      generate an error
    endif
    addr_op = GenImmediate(CDFG, addr, true)
    mark addr_op as being a fixup node
  else if immediate offset is 0 (handling split field if required) then
    // if there is an offset of 0 then the base register can be used
    // directly
    addr_op = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
  else
    // We have an immediate offset - this is translated into an explicit
    // add operation between the immediate and the base register. The
    // offset is negated if this is a down offset
    immmed_op = GenImmediate(CDFG, immediate value negated if U bit reset
                             (handling split field if required), false)
    base_op = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
    addr_op = GenBinaryOp(CDFG, addition unit, add, base_op, immmed_op)
  endif
else if shift value is 0 (i.e. two register ops) or
    half word/signed byte access then
  // we have two register offsets - these are combined with an addition

```

```

// or subtraction as required
op1 = GenerateRegRead(Rn operand)
op2 = GenerateRegRead(Rm operand)
if U bit is reset (i.e. subtraction) then
    addr_op = GenBinaryOp(CDFG, addition unit, add, op1, op2)
else
    addr_op = GenBinaryOp(CDFG, subtraction unit, subtract, op1, op2)
endif
endif
else
    // we have a register offset that is shifted by a fixed amount - code
    // for an immediate shift is generated
    op2 = GenShiftImmed(CDFG, RegDefs, shift type from Sh field,
                        Rm operand, shift amount, false, FuncType)
    op1 = GenRegRead(CDFG, RegDefs, Rn operand, FuncType)
    if U bit is reset (i.e. subtraction) then
        addr_op = GenBinaryOp(CDFG, addition unit, add, op1, op2)
    else
        addr_op = GenBinaryOp(CDFG, subtraction unit, subtract, op1, op2)
    endif
endif
endif

return addr_op

```

### 9.4.13 Immediate Shift Generation

```

Node GenShiftImmed(CDFG, RegDefs, ShiftType, SourceReg, ShiftAmount,
                  GenFlags, FuncType)
// Generates a shift operation by a fixed amount in an immediate value.
// Uses the byte and bit shifters back-to-back to accomplish the full
// shift. Writes the carry flag if required. Returns the data producing
// operation.
// RegDefs is the current register definitions
// ShiftType is the type of shift to be performed
// SourceReg is the register holding the data to be shifted
// ShiftAmount is the fixed amount to be shifted
// GenFlags is true if the carry flag should be written back
// FuncType is the type of the function being scheduled

// read the data to be shifted
data_op = GenRegRead(CDFG, RegDefs, SourceReg, FuncType)

// if the shift is greater than 7 then a byte shift is required
if ShiftAmount requires a byte shift (greater than shift of 7) then
    amount_op = GenImmediate(CDFG, byte shift amount, false)
    data_op = GenBinaryOp(CDFG, Byte Shift Unit, ShiftType, data_op,
                        amount_op)
endif

// if the shift amount is not a multiple of 8 then a bit shift is required -
// this may require a link to any earlier byte shift in order to produce the
// carry flag
if ShiftAmount requires a bit shift (not multiple of 8) then
    amount_op = GenImmediate(CDFG, bit shift amount, false)
    data_op = GenBinaryOp(CDFG, Bit Shift Unit, ShiftType, data_op, amount_op)
endif

// write the carry flag if required
if GenFlags then
    RegWrite(CDFG, RegDefs, Register holding C flag, data_op, carry output
            port, FuncType)
endif

return data_op

```

### 9.4.14 Register Shift Generation

```

Node GenShiftReg(CDFG, RegDefs, ShiftType, SourceReg, ShiftReg, GenFlags,
                FuncType)
// Generates a shift operation by a variable amount as stored in a register.
// Uses the byte and bit shifters back-to-back to accomplish the full
// shift. Writes the carry flag if required. Returns the data producing
// operation.
// RegDefs is the current register definitions
// ShiftType is the type of shift to be performed
// SourceReg is the register holding the data to be shifted
// ShiftReg is the register holding the amount to be shifted
// GenFlags is true if the carry flag should be written back
// FuncType is the type of the function being scheduled

// load the shift amount and the data to be shifted
amount_op = GenRegRead(CDFG, RegDefs, ShiftReg, FuncType)
data_op = GenerateRegRead(CDFG, RegDefs, SourceReg, FuncType)

// perform a byte shift followed by a bit shift - may need additional
// output from byte shift to bit shift for the carrt
data_op = GenBinaryOp(CDFG, Byte Shift Unit, ShiftType, data_op,
                    amount_op)
data_op = GenBinaryOp(CDFG, Bit Shift Unit, ShiftType, data_op, amount_op)

// if flags should be generated then they come from the carry output of.
// the bit shift operation (this must reproduce the carry out from the
// byte shift if there is no bit shift)
if GenFlags then
    GenRegWrite(CDFG, RegDefs, Register holding C flag, data_op, carry output
                port, FuncType)
endif

return data_op

```

### 9.4.15 Condition Evaluation Generation

```

void GenCondEval(CDFG, RegDefs, ConditionType, FuncType)
// Evaluates a particular condition and generates a squash operation based
// on that condition. This is used to handle condition instructions. The
// condition is evaluated and the body of the instruction is put into a
// following strand whose execution is controlled by the squash. The
// condition is determined from a table depending on the ConditionType
// that loads the required individual architectural flag registers.
// ConditionType gives the type of condition as a 4 bit ARM format
// condition field

// a chain of operations is performed to evaluate the condition from the
// individual condition registers
if condition is not always then
    if condition is never then
        // generate a known false state
        flag_state = GenImmediate(CDFG, 0, true)
    else
        // the first condition from a table is loaded
        first_load = initial operation required for condition
        flag_state = GenRegRead(CDFG, RegDefs, register required for
                                first_load, FuncType)

        // the subsequent conditions are combined as required
        foreach load_req for the ConditionType do
            combine_state = GenRegRead(CDFG, RegDefs, register required for
                                        combination, FuncType)
            flag_state = GenBinaryOp(CDFG, Logical Unit, AND, OR or XOR as

```

```

        required by state table, flag_state, combine_state)
    if state table requires an inversion then
        flag_state = GenUnaryOp(CDFG, Logical Unit, NOT, flag_state)
    endif
endfor
endif

// The actual squash operation is generated - it is made the current
// squash operation for the current strand. This allows subsequent
// branch operations to determine which squash controls their execution -
// this is used if a branch is deleted if the destination is internal to
// the region
squash_op = GenUnaryOp(CDFG, squash unit, true or inverse depending
                      on required flag state, flag_state)
mark squash_op as the squash instruction for the current strand - this may
be carried into subsequent strands
endif

```

### 9.4.16 PC Update Generation

```

Node GenPCUpdate(CDFG, RegDefs, InstrAddr, FuncType)
// Updates the value of the PC architectural register for the
// current source instruction. This update is performed for each
// source instruction. For main execution code these updates are
// optimised away. They are maintained for the shadow code to
// keep the PC value up to date and to provide the input to
// the instruction predicate unit (which is able to perform
// breakpointing). The PC value is formed by loading it with an
// immediate value of the original instruction address.
// Returns the operation that produces the updated PC value.
// RegDefs is the current state of the register definers
// InstrAddr is the address of the current instruction
// FuncType is the type of the function being scheduled

// load the PC value as an immediate value
instr_addr = GenImmediate(CDFG, InstrAddr, true)
mark instr_addr as requiring a fixup

// write the current PC value into the PC register
GenRegWrite(CDFG, RegDefs, PC register, instr_addr, main port)

return instr_addr

```

### 9.4.17 Instruction Predicate Generation

```

void GenInstrPredicate(CDFG, PCUpdateNode)
// Instruction predicate nodes are used to guard all original instructions.
// An instruction predicate is generated just before the operations
// associated with a particular original instruction. These
// are always optimised out of the main code but remain present for the
// shadow code. They access the breakpoint comparison unit to compare
// original PC values. This provides a predicate to guard all subsequent
// execution of the strand and an interrupt is generated upon a match
// to invoke the monitor to handle the breakpoint. The instruction
// predication is a dependent upon all operations in the previous
// instruction of the strand. The predicate instruction is a
// dependee of all operations for the following original instruction.
// PCUpdateNode is the node generating the updated PC value for the
// instruction about to be translated

pred_op = GenUnaryOp(CDFG, instruction predicate unit, PCUpdateNode)
foreach prev_node in the CDFG do
    if prev_node != pred_op and prev_node is in the current strand then

```

```

        create a strong control arc from prev_node to pred_op
    endif
endfor
add pred_op as the current instruction predicate for the CDFG (this
    is then added as a predecessor for subsequent instructions)

```

#### 9.4.18 Register Read Generation

```

Node GenRegRead(CDFG, RegDefs, SourceReg, FuncType)
// Generates a read of an architectural register. The register definition
// information is used to generate dependencies to the write(s) that
// generate the value in the register. There may be multiple writers if
// the control flow is confluenced. These dependencies prevent the read
// operation being issued before the write. Returns the read node.
// RegDefs is the current state of the register definitions
// SourceReg is the architectural register to be read
// FuncType is the type of the function being scheduled

if reg_num is the PC register then
    // reading from the PC register is a special case - we convert it into
    // an immediate load that is fixed up in a later pass to give the correct
    // value even if the code is relocated
    read_op = GenImmediate(CDFG, PC value of instruction + 8, true)
    mark the read_op as a fixup node
else
    // read the appropriate register - the register file may be partitioned
    // into statically to improve parallelism
    reg_unit = the appropriate register unit depending on SourceReg
    reg_num = the appropriate register number for the selected unit based on
        SourceReg
    read_op = GenOp(CDFG, reg_unit, reg_num + read selection bit, empty)

    // generate any required dependencies to earlier writes of the register
    foreach prev_def that is a previous definition in RegDefs do
        if prev_def is in the same strand or FuncType = Critical then
            // a strong ordering is required
            if there is only one previous definition of the register then
                // if there is only a single reaching definition then it is
                // marked as a tunnel arc for potential later optimisation
                generate tunnel arc from prev_def to read_op
            else
                // a strong arc ensures that the read is not issued before
                // any of the earlier writes
                generate a strong control flow arc from prev_def to read_op
            endif
        else
            // a weak arc is generated that aborts the reading strand if the
            // dependency is violated
            generate a weak control flow arc from prev_def to read_op
            generate a corresponding conditional arc from prev_def to the
            guard operation associated with the current strand
        endif
    endfor

    // generate an information arcs to any first use of the register in the
    // strand - this may be used later to rewrite the graph to only read
    // the register once in the strand
    if there is a first use of the register in RegDefs then
        generate an information arc from the first use to read_op
    else
        make read_op the first use of the register in RegDefs
    endif
endif

return read_op

```

### 9.4.19 Register Write Generation

```

void GenRegWrite(CDFG, RegDefs, DestReg, DataSource, DataPort, FuncType)
// Generates a write to an architectural register. This generates the write
// operation and also updates the register definition information. This is
// used to keep a record of which nodes have updated the architectural
// registers in order to maintain dependencies between them. Control arcs
// are generated between writes to the same register in the original program
// order within a strand. This ensures that live out register values are
// preserved by keeping the same order of writes to architectural registers.
// Arcs to writes in preceding strands are made weak unless the CDFG being
// constructed is critical. A conditional arc is constructed to the guard
// operation for the strand - so that if the register write order is
// violated then previous strands must commit before committing the current
// one. This function also generates code for return, indirect function
// calls and switch branches. These are instruction that modify the PC
// value.
// RegDefs is the current state of the register definitions
// DestReg is the architectural register that is to be written
// DataSource is the node that generates the data to be written
// DataPort is the port of the DataSource where the data is available
// FuncType is the type of the function being scheduled

// perform the actual register write - the architectural registers may be
// partitioned statically between a number of individual register units
reg_unit = the appropriate register unit depending on DestReg
reg_num = the appropriate register number for the selected unit based on
           DestReg
param = build parameter with DataSource and DataPort
write_op = GenOp(CDFG, reg_unit, reg_num + write selection bit, param)

// if a register is written then the first use information is cleared
clear the first use entry associated with the register

// generate the required dependency arcs to previous writes of the same
// architectural register to maintain the write order
if there is a previous list of definitions for the register in RegDefs then
    foreach prev_def that is a previous definition do
        if prev_def is in the same strand or FuncType = Critical then
            // strong write order is maintained in the same strand or in
            // critical functions
            generate a strong control flow arc from prev_def to write_op
        else
            // we generate a weak arc to the previous write and a
            // conditional arc to the guard for the current strand - thus
            // if the write order is violated then the current strand is
            // aborted unless it is the first being executed
            generate a weak control flow arc from prev_def to write_op
            generate a corresponding conditional arc from prev_def to the
            guard operation associated with the current strand
        endif
    endfor
endif

// any write to the PC register is actually a branch in disguise - these are
// used to implement indirect calls, returns and branches for switch code
if DestReg is the PC register then
    // if this is a special function type then there may be particular
    // register restores required for a return
    if instruction is not marked as an indirect call then
        if FuncType = SlowInterrupt or FuncType = FastInterrupt then
            // if this is an interrupt function then we need to generate for the
            // code to reload the volatile registers
            foreach volatile register, stack pointer and condition code reg do
                generate code to reload register into main register file
            endfor
        endif
    endif
endif

```

```

    endfor
    else if FuncType = SWIHandler then
        // 'if this is a return from a SWI handler then we need to restore
        // the appropriate registers
        restore the condition code registers
        restore the link registers from r14_svc
        restore the stack pointer from r13_svc
    endif
endif

// The new PC value will point to an indirect link that will actually hold
// the address to be returned to. We generate an operation to load it. In
// the case of a return the link is stored just after the call site. In
// the case of an indirect call it is in the first word of the function to
// be called. In the case of a branch for switch code each possible
// destination has a link.
load_op = GenOp(CDFG, memory unit, load, param)
GenBranch(CDFG, load_op)
endif

// the register definitions need to be updated with the new write
// overwrite the current value of the register in RegDefs with the write_op

// if we have modified any of the condition code registers responsible for
// the current strand condition then we need to clear the current
// condition - this handles conditional operations that actually update the
// condition codes themselves so a subsequent instruction with the same
// condition guard needs a new strand
if DestReg is one of the condition code registers and the condition code
    is relevant to the entry condition of the strand then
    clear the entry condition of the strand
endif

```

## 9.4.20 Immediate Value Generation

```

Node GenImmediate(CDFG, ImmedValue, StrandReq)
// Generates an operation to load the given immediate value. All immediates
// are loaded in a single operation. A number of immediate units are
// made available. The narrowest possible immediate unit is chosen.
// The operation node in the CDFG is returned.
// StrandReq is true if an RSN is required as part of the immediate field -
// this is the case if the immediate feeds into the first operand of its
// consumer

foreach immed_unit from narrowest to widest do
    if ImmedValue can be represented in the width of immed_unit and
        RSN field is present if StrandReq is asserted then
        // generate an operation to load the immediate value - the method
        // port specifies the value for the immediate unit
        imm_op = GenOp(CDFG, immed_unit, ImmedValue,
            empty (no other parameters))
        return imm_op
    endif
endforeach

// all immediate values can be represented - other than those using an ARM
// shift into the most significant bits but those are handled at a higher
// level
report an error

```

## 9.4.21 Branch Generation

```

void GenBranch(CDFG, DestPCValue)

```

```
// Generates a branch operation with the given original destination PC
// value. The branch may implement a call, original branch in the code
// or a call to the handler for a software interrupt. The squash node
// that controls the execution of the current strand is stored so that
// if the branch is later deleted (because it turns out the destination
// is inside the region) then the squash operation can be updated
// appropriately
// DestPCValue is the destination of the branch in terms of the original
// PC value

// Generate the destination address - this is a full destination
// including strand number and address. It is handled as a fixup
// node so that it is calculated after the binary has been generated to
// allow differencing values without forcing re-scheduling of a function.
fixup_node = GenOp(CDFG, Wide Immediate Unit, DestPCValue)
mark fixup_node as being a fixup node

// the branch operation itself is generated
branch_node = GenUnaryOp(CDFG, branch unit, initiate branch, fixup_node)
store information about the squash node that controls the execution of
the current strand
```

#### 9.4.22 Unary Operation Generation

```
Node GenUnaryOp(CDFG, UnitType, UnitMethod, ParamNode)
// Generates an operation requiring a single parameter in addition to
// the method
// UnitType is the type of unit required
// UnitMethod is the method to be executed on the unit
// ParamNode is the operation which is the source of the parameter - the
// main result port is used

param_list = build parameter from ParamNode main result port
return GenOp(CDFG, UnitType, UnitMethod, param_list)
```

#### 9.4.23 Binary Operation Generation

```
Node GenBinaryOp(CDFG, UnitType, UnitMethod, LeftNode, RightNode)
// Generates an operation requiring a two parameters in addition to
// the method
// UnitType is the type of unit required
// UnitMethod is the method to be executed on the unit
// LeftNode is the operation which is the source of the first parameter -
// the main result port is used
// RightNode is the operation which is the source of the first parameter -
// the main result port is used

param_list = build parameter from LeftNode main result port
param_list = add build parameter from RightNode main result port
return GenOp(CDFG, UnitType, UnitMethod, param_list)
```

#### 9.4.24 Operation Generation

```
Node GenOp(CDFG, UnitType, UnitMethod, ParamList)
// Creates a new operation and adds it to the CDFG. A list
// of parameters is specified (that may be empty) which show
// the operands of the operation.
// UnitType is the type of unit required
// UnitMethod is the method to be executed on the unit
// ParamList is the list of required parameters - each has a
// source node and the port from the node on which the value
```

```

// is available

new_node = create a node of the required UnitType with UnitMethod
            in the CDFG
if UnitMethod cannot be issued speculatively then
    // If the operation cannot be issued speculatively then the
    // phase barrier for the strand is made to depend on it to ensure
    // it is only issued during the commit phase. The arc indicates
    // the number of clock cycles to resolve a commit.
    create control flow arc from current strand phase barrier to
    new_node with a latency for commit resolution
endif

// all emitted operations depend on the most recent instruction
// predicate
create control flow arc from the instruction predicate for the
current instruction to new_node

// create the data inflows to the operation as data arcs
foreach operand in ParamList do
    create a data arc from the source node and port in ParamList - the arc
    is labelled with the latency of results from the unit (also add
    the relevant transport costs - from transport delay in node)
endfor

// Each method may be a member of a number of ordering sets. All methods
// within each set must maintain the same ordering as in the original
// program. This allows dependencies between operations to be taken into
// account. Control arcs are generated to ensure the same ordering is kept.
// Virtual registers in RegDefs (not actually shown as accessible here)
// are used to keep the ordering information. The confluence operations
// performed on RegDefs also confluence the ordering sets.
foreach order_set that the UnitMethod is a member of do
    generate a control arc to the last instruction issued in set (actually
    use virtual registers in RegDefs that needs to be made accessible to
    this routine)
    add new_node to the appropriate virtual register in RegDefs
endfor

return new_node

```

### 9.4.25 Memory Dependency Addition

```

void AddMemoryDependencies(CDFG, NewNode, FuncType)
// Adds memory dependencies in the CDFG for the NewNode. Looks at all
// previous store nodes that are in strands that are reachable from the
// current one (or are in earlier parts of the same strand). Generates
// dependency arcs as required. Also generates chaz operation nodes
// as appropriate to allow loads to migrate earlier than potentially
// aliased stores. Memory dependencies on call instructions are not
// required as operations cannot be migrated earlier than a call as
// the return causes a region restart.
// NewNode is the load or store node for which dependencies to earlier
// memory accesses need to be added
// FuncType is the type of the function being scheduled

foreach prev_node in the CDFG do
    if prev_node is in a reachable strand from current one in CDFG then
        if prev_node is a store then
            // the node is a store and is reachable from the current strand so
            // we determine if they might be aliased
            dep_req = none
            alias = AliasCheck(prev_node, NewNode)
            if alias = PartialAlias or alias = FullAlias then
                // there is a definite alias
                if prev_node is in the current strand or FuncType = Critical or

```

```

                                NewNode is a store then
                                // dependencies in the current strand, dependencies in a
                                // critical function and store to store dependencies must be
                                // strongly ordered
                                dep_req = Strong
                            else
                                // if the store-load are issued out of order then we must
                                // abort the strand holding the load
                                dep_req = CondAbort
                            endif
                        else if alias = MaybeAlias then
                            // the accesses may be aliased
                            if prev_node is in the current strand or NewNode is a store then
                                // store-store dependencies or dependencies within the same
                                // strand are strongly ordered even if the alias is only
                                // potential
                                dep_req = Strong
                            else
                                // if the store-load are issued out of order then we must issue
                                // a check hazard operation to abort the load strand if there
                                // was an actual alias
                                dep_req = CondChaz
                            endif
                        endif
                    endif

                // generate the dependency as required
                if dep_req = Strong then
                    // we have a strong alias - if a load is definitely loading
                    // data from an earlier store then a tunnel arc is generated that
                    // may be used by the CDFG optimiser to eliminate the load
                    // operation
                    if alias = FullAlias and NewNode is a load then
                        create a tunnel arc between prev_node and NewNode
                    else
                        create a control arc between prev_node and NewNode
                    endif
                else if dep_req = CondAbort then
                    // we generate a weak arc that aborts the load strand if it is
                    // violated
                    create a weak control arc between prev_node and NewNode
                    create a corresponding conditional arc between prev_node and the
                    guard operation for the current strand
                else if dep_req = CondChaz then
                    // we create the chaz operation and put it into the speculative
                    // phase of the current strand
                    chaz_node = a new chaz operation node
                    add the chaz_node to the CDFG and make it dependent on the phase
                    phase barrier for the current strand (so that it has to be
                    issued before it)

                    // the store-load weak dependency arc is created - a conditional
                    // arc to the chaz operation is activated if the dependency order
                    // is violated
                    create a weak control arc between prev_node and NewNode
                    create a corresponding conditional arc between prev_node and
                    chaz_node

                    // we get the nodes generate the store and load addresses and
                    // route them to the chaz operation as well - the load address
                    // is made conditional since it is an acycle arc (the load
                    // may be forced into the committed phase of the current strand
                    // by instruction predicates but the chaz must be in the
                    // speculative phase)
                    st_addr_node = the node generating the address for the store
                    ld_addr_node = the node generating the address for the load
                    create a data flow arc from st_addr_node to the chaz_node
                    create a conditional data flow arc from ld_addr_node to the
                    chaz_node (this is created as a conditional node as it is

```

```

        acyclic)
    endif
endif
endif
endif
endfor

```

## 9.4.26 Memory Alias Checking

```

enum AliasType {
    NoAlias,           // the operations are definitely not aliased
    MaybeAlias,        // the is not enough information to tell if the accesses
                        // are aliased
    PartialAlias,      // the operations definitely overlap but only partially
    FullAlias}         // there is full overlap between the operations and they
                        // are of the same size

AliasType AliasCheck(PrevNode, CurNode)
// Performs an alias check between two nodes in the CDFG. Each of the nodes
// should be a load or store operation.
// PrevNode is the earlier memory access node
// CurNode is the later memory access node

if PrevNode and CurNode both use addresses calculated from additions and
PrevNode and CurNode both have immediate offsets for the additions and
PrevNode and CurNode both use the same base register (generated from
the same node) then
    // both accesses are immediate offsets from the base address - this
    // will catch most stack accesses and also accesses to the same
    // structure or class
    prev_offset = offset for PrevNode
    cur_offset = offset for CurNode
    if prev_offset and cur_offset are identical as are the access sizes then
        // the accesses are completely aliased
        return FullAlias
    else if prev_offset and cur_offset could overlap given access sizes then
        // the accesses are partially overlapped
        return PartialAlias
    else
        // the accesses are definitely not aliased
        return NoAlias
    endif
else if PrevNode and CurNode both have the same address generator then
    // this handles cases where the address does not come from an addition but
    // they both have exactly the same base register
    if PrevNode and CurNode both have the same size then
        // it is a full alias if the access sizes are also identical
        return FullAlias
    else
        // if the access sizes are not identical then the overlap is only
        // partial
        return PartialAlias
    endif
endif

// we do not have enough information to determine if the accesses are
// aliased or not
return MaybeAlias

```

**This Page is Inserted by IFW Indexing and Scanning  
Operations and is not part of the Official Record**

**BEST AVAILABLE IMAGES**

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☒ FADED TEXT OR DRAWING
- ☐ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☒ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☐ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☐ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: \_\_\_\_\_

**IMAGES ARE BEST AVAILABLE COPY.**

**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**